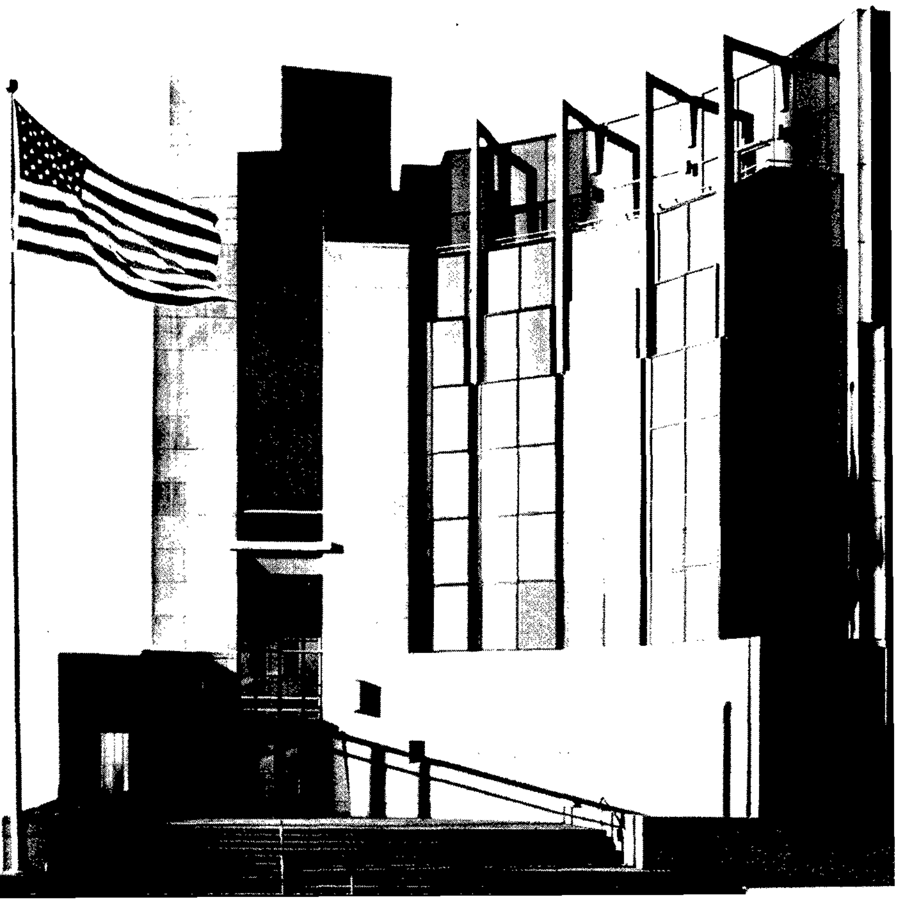
Carnegie Mellon
Software Engineering Institute

Pin Component Technology (V1.0) and Its C Interface

Scott Hissam
James Ivers
Daniel Plakosh
Kurt C. Wallnau

April 2005

TECHNICAL NOTE
CMU/SEI-2005-TN-001



Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Pin Component Technology (V1.0) and Its C Interface

CMU/SEI-2005-TN-001

Scott Hissam
James Ivers
Daniel Plakosh
Kurt C. Wallnau

April 2005

Predictable Assembly from Certifiable Components Initiative

This work is sponsored by the U.S. Department of Defense.

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

Abstract

Pin is a basic, simple component technology suitable for building embedded software applications. Pin implements the container idiom for software components. Containers provide a prefabricated "shell" in which custom code executes and through which all interactions between custom code and its external environment are mediated. Pin is a component technology for pure assembly—systems are assembled by selecting components and connecting their interfaces (which are composed of communication channels called pins).

This report describes the main concepts of Pin and documents the C-language interface to Pin V1.0.

1 Introduction

A *component technology* comprises a component model and a runtime environment [Bachmann 00]. The component model defines the logical and implementation structure of components and applications constructed from components, and defines rules for how components may interact with one another and how they share resources. The runtime environment enforces these rules of interaction and provides basic services for resource sharing, communication, scheduling, and the like.

The Pin component technology is based on an earlier component technology developed by the Carnegie Mellon® [1] Software Engineering Institute (SEI) for the Environmental Protection Agency (EPA) [Plakosh 99]. Pin has since been further developed for use in prediction-enabled component technologies (PECTs) [Wallnau 03b]. Pin is a basic, simple component technology. A component technology for the class of systems we are targeting—embedded time- and safety-critical systems—should be small with an implementation that is (relatively) transparent; in fact, a large and opaque component technology would be counterproductive for this class of applications. There have been several generations of Pin implementations, and we expect more generations to follow.

About This Report

The objective of this report is to describe the logical structure of the Pin component technology and to document the application programming interface (API) for a version of Pin used in PECTs for substation automation [Hissam 03] and industrial robotics [Hissam 04a]. This report does not intend to propose a standard interface to Pin or any other component technology. In particular, we expect this API to undergo changes, some of which are already in development.

The intended audience for this report is the practitioner who is interested in understanding or developing software component technologies for embedded applications. Although this report does not give specific guidance on how to develop such a component technology, it does document a component technology that has been useful in nontrivial settings. Further insight into the design of Pin is provided in reports on the construction and composition language (CCL) [Ivers 02], [Wallnau 03a]; CCL is an architecture description language in the "component and connector" style that has been further specialized to work with the Pin component technology.

1. Carnegie Mellon is registered in U.S. Patent and Trademark Office by Carnegie Mellon University.

The logical structure of Pin is described in Section 2. Our current work on Pin is described in Section 3. Appendix A describes the Pin application programming interface (API), while Appendix B provides a simple but illustrative example Pin application.

2 Overview of Pin

2.1 Major Design Objectives

The design of Pin is governed by five overall design objectives that transcend requirements imposed by particular applications and that will continue to govern the evolution of Pin. In brief, Pin should

1. have a simple programming model and an execution model that supports the semantics of UML statecharts
2. provide various ways to enforce extrinsic (to Pin) design and implementation constraints
3. introduce only the most basic features needed for building predictable embedded software
4. be adaptable to the needs of new applications and platforms
5. be freely distributable

The first objective is important for reasons of usability but is even more important for the purpose of automation. In particular, a simple programming model makes automated code generation straightforward and makes the code generators themselves relatively immune from changes to the Pin implementation. Further, specifications of component behavior, and the formal basis for their analyses, can build on a widely available specification language, UML. The second objective provides the flexibility needed to integrate new reasoning frameworks [Bass 05] into Pin-based PECTs; new reasoning frameworks may have assumptions that we wish to preserve as invariants in systems built using Pin. The third objective appeals to the benefits of parsimony: the simpler the implementation, the less chance it will introduce unanticipated runtime effects. The fourth objective recognizes that the needs of a broader range of applications and platforms can be supported without compromising a simple programming model by identifying key variation points, such as interaction mechanisms and scheduling policies. The fifth objective reflects our ultimate desire to make the results of our work widely available to practitioners, students, and researchers.

2.2 Overall Structure

Pin implements features of component technology that are frequently encountered in research prototypes as well as commercial products:

- Pin implements the container idiom for software components. Containers provide prefabricated "shells" in which custom code executes. All interactions between the custom code and its external environment are mediated by the container, which may impose container-type-specific coordination policies. A component is a container and its custom code.
- Pin components are fully encapsulated. The container ensures that custom code can interact with its environment only through container-mediated interfaces. While full encapsulation "feels" restrictive to developers accustomed to unfettered access to the runtime environment, the results are systems with fewer hidden component dependencies and fewer changes for unanticipated component interactions.
- Components are independently deployable binary implementations with explicit context dependencies (a widely accepted starting point, popularized by Szyperski [Szyperski 02]). Each Pin component is implemented as a distributable dynamic link library (DLL). Since components are fully encapsulated, all environmental dependencies (on the runtime environment or other components) are fully explicit.
- Pin supports a model of pure assembly. Applications are constructed by connecting components using a repertoire of connectors. Each connector may impose coordination policies beyond those provided by containers; for example, queuing policies on message buffers. Assembly is "pure" because it is declarative; point-to-point custom interaction code (a.k.a. "glue" code) is not permitted.[2]
- A component runtime environment provides services and enforces component interaction policies. Services include access to the underlying platform; for example, timers, interrupts, and input devices. Interaction policies governing shared resources, such as process scheduling and intercomponent communication, are also provided by the runtime. Lastly, the runtime provides a portability layer for components and their assemblies.

The overall logical structure of Pin is depicted in Figure 1.

2. The effects of custom glue code can be simulated, of course, by encapsulating it as a Pin component.

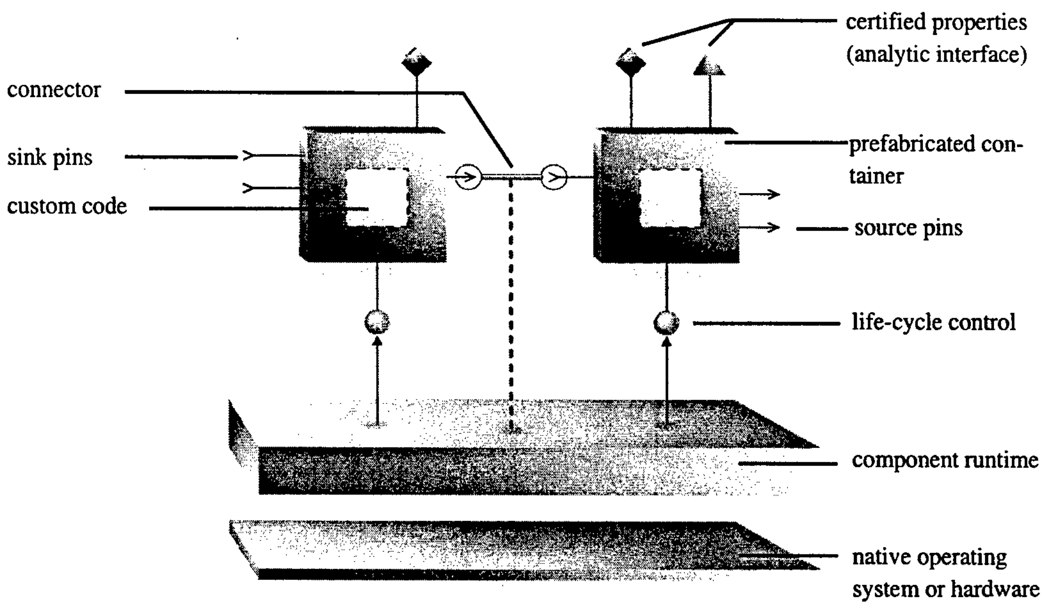


Figure 1: Major Structure of the Pin Component Technology

Pin V1.0 has a number of limitations. Some are "principled restrictions" that reflect our underlying design philosophy, as well as the special considerations of our intended application domain (embedded, safety- and performance-critical software). Other limitations arise only from expediency and will likely be relaxed in future implementations. It is not always clear whether a limitation is a principled restriction or an expediency, and the interplay of design objectives 3 and 4 almost requires a flexible boundary between those two things.

Without prejudice to classification, the following summarizes the main limitations of Pin V1.0:

- Assembly topologies are fixed: we assume a closed world with static configurations.
- Distributed and hierarchical assembly (i.e., assemblies of assemblies) is not supported.
- Each component reaction (see Section 2.3) has a thread of control; the notion of unthreaded reactions (as outlined by Ivers, Sinha, and Wallnau [Ivers 02]) is not supported.
- Message sizes for intercomponent communication are fixed, and queue lengths for connectors are fixed (but statically configurable).
- Only two types of connectors are supported—for synchronous and asynchronous interaction (although different connector implementations can be statically configured).
- Only one type of container is provided (although more are currently under development for the next version of Pin).

2.3 Components

A Pin component consists of two parts: (1) a user-supplied (custom) function and (2) the Pin-supplied (prefabricated) container function. The interface structure of a Pin component is displayed in Figure 2.

User-supplied code must conform to an interface contract defined by the container ("user code API" in Figure 2); the container uses this interface to invoke (via the "user code plug-in") user-supplied code in response to requests from the Pin runtime environment or from other components. Conversely, the container provides an interface ("container API") that custom code uses to requests service from the runtime environment or from other components. The container also presents a single interface to the environment ("component API"), which conforms to an environment-defined plug-in interface (not depicted). The container provides two additional plug-in interfaces ("directory server plug-in" and "IPC plug-in")[3] to make it possible to deploy components to different runtime environments, possibly having different intercomponent (or interprocess) communication mechanisms. Pin components are packaged and deployed as dynamic link libraries (DLLs).

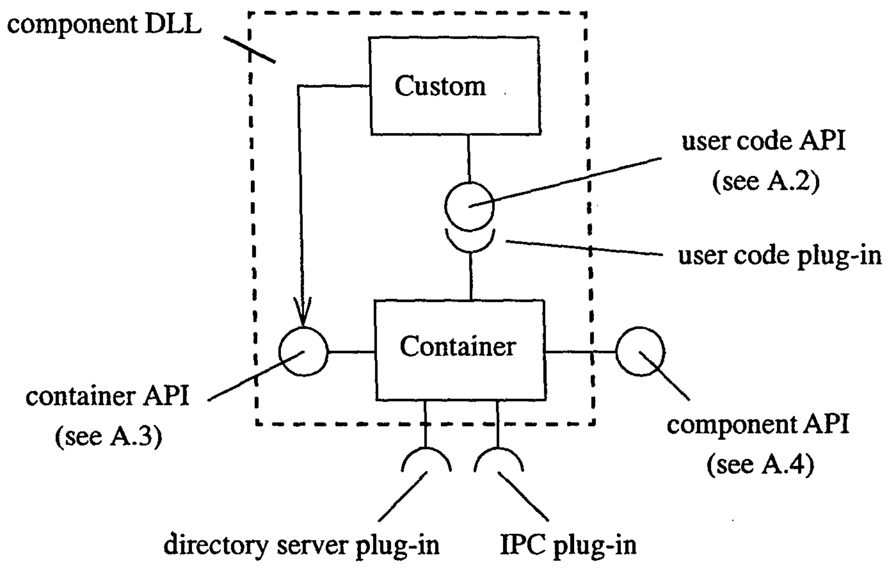


Figure 2: Pin Component and Container Contracts

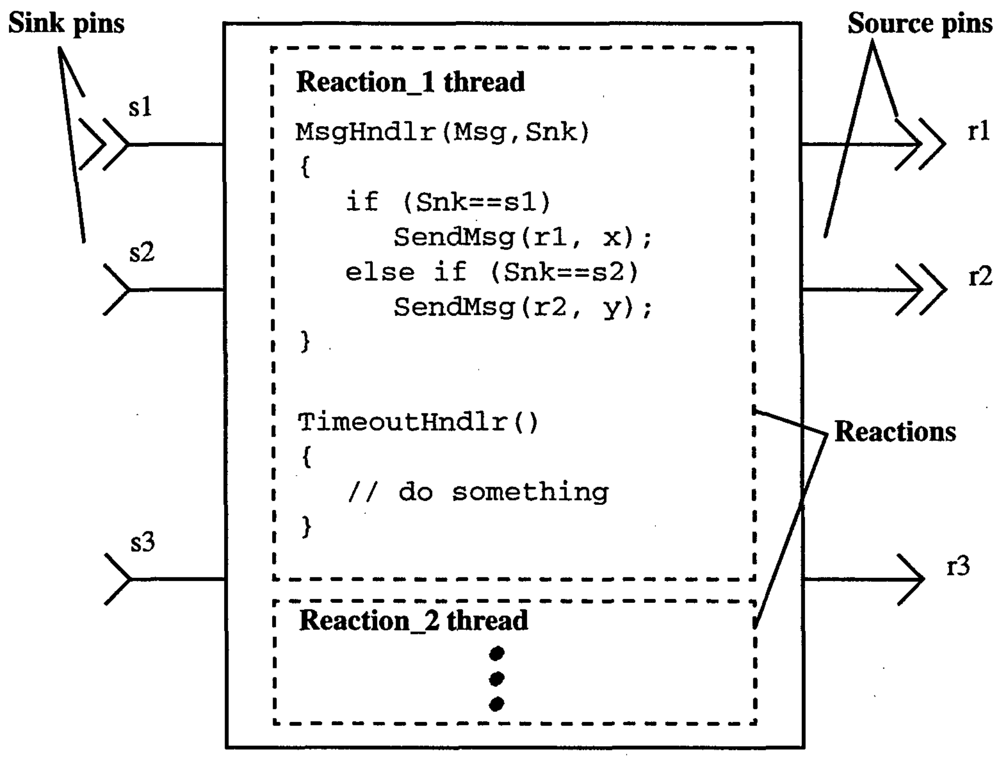
The user code API and container API are quite simple, each defining fewer than ten operations. The user code API defines operations for creating, destroying, and initializing runtime instances of the component. The container API provides operations for sending requests for

3. IPC stands for interprocess communication.

service to the environment. The simplicity of these interfaces reflects our concern to make the programming model for Pin as simple as possible.

In practice, the isolation of the custom code from its environment is not as complete as depicted in Figure 2; there are inevitable dependencies on standard runtime libraries. These dependencies could be problematic if external libraries violate reasoning framework assumptions. For example, an external library might introduce a source of unbounded priority inversion, which in turn would invalidate the predictions of a performance theory. There are a variety of ways to enforce strict isolation. (This is a topic of ongoing SEI research.)

The logical structure of a Pin component (in particular, the custom code part of a Pin component) is depicted in Figure 3. Wallnau and Ivers provide a more detailed description of the concepts [Wallnau 03a].



```
MsgHndlr(Msg,Snk)
{
   if (Snk==s1)
      SendMsg(r1, x);
   else if (Snk==s2)
      SendMsg(r2, y);
}

TimeoutHndlr()
{
   // do something
}
```

Figure 3: The Logical Structure of a Pin Component: Pins and Reactions

The custom code of a Pin component is organized as a set of one or more *reactions*. The container creates a thread for each reaction.[4] This thread waits for the arrival of stimulus on a sink

4. The "unthreaded reaction" feature of CCL was not implemented in Pin V1.0.

pin; stimulus is FIFO queued. On receipt of stimulus, user code is executed; when this processing is complete, control is returned to the container, which then checks for further stimulus, *ad infinitum.*

Each reaction accepts stimuli from *one or more* sink pins and produces responses on *zero or more* source pins. Each sink pin provides stimulus to *at most* one reaction; each source pin is used by *at least* one reaction. Pins support synchronous and asynchronous interaction. Synchronous interaction has the familiar procedure-call semantics (although it is implemented using a messaging system), while asynchronous interaction has the familiar event-based semantics.

2.4 Assemblies

Component instances receive stimulus through sink pins and respond through source pins; enabling interaction among component instances requires that we connect a source pin of one component instance to the sink pin of another. An assembly of components consists of a static topological arrangement of component instances.

Pin V1.0 does not explicitly support the notion of assembly. In particular, there is no notion of *assembly container* (although this will change in a future release). Instead, we construct assemblies implicitly as a main (top-level) executable program that manages the component and (implicit) assembly life cycle. Typically, assembly programs (also called assembly "controllers" in the Pin vernacular) are generated automatically from CCL specifications. The life cycle of an assembly controller is shown in Figure 4.

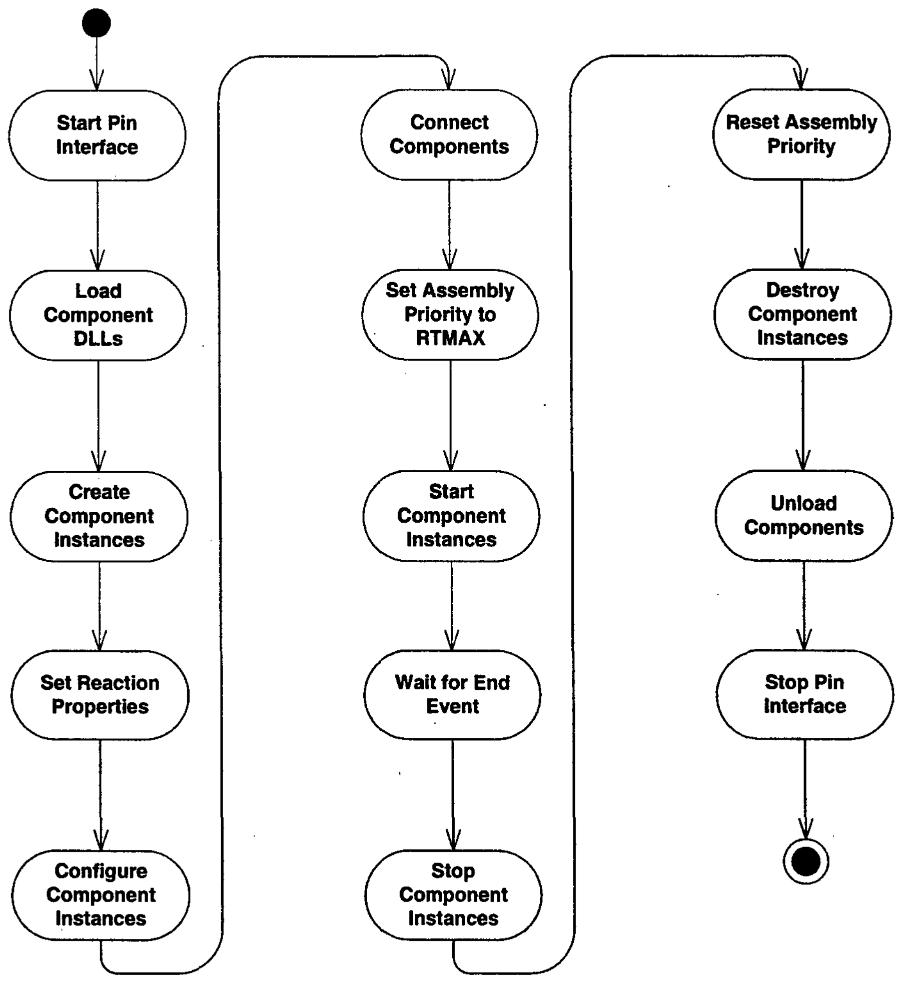


Figure 4: The Assembly Controller Life Cycle

There is no intrinsic reason why component instances in different controllers can't be connected (indeed, we use a directory server to look up component interface details, and the connector mechanisms supported by the Pin runtime are UDP based). However, we will not consider Pin as supporting distributed assembly (or uniprocessor hierarchical assembly) until such concepts are explicitly supported via assembly containers or their equivalents.

2.5 Runtime Environment

As discussed in earlier reports (e.g., in the work of Bachmann and colleagues [Bachmann 00]), a component runtime environment plays a role with regard to component assemblies that is analogous to the role operating systems play with regard to processes. The analogy is a strong one, and, not surprisingly, the boundary between component runtime and operating system is fuzzy and, to some extent, arbitrary.

The component runtime for Pin V1.0 is a pragmatic amalgamation of services needed to support prototypes built for electric grid substation automation [Hissam 03] and industrial robot control [Hissam 04a]. Only the major elements of the Pin V1.0 runtime are shown in Figure 5.

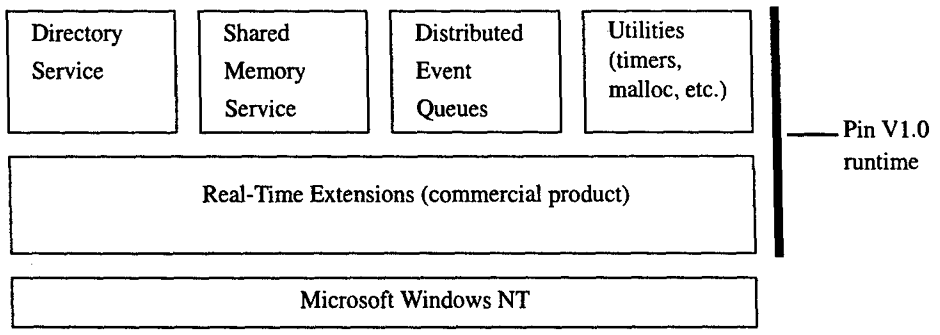


Figure 5: Pin V1.0 Runtime Environment

There are two layers of service in the runtime. The bottom layer provides real-time thread support and is implemented by a commercial product that provides these services as extensions[5] of the Microsoft Windows NT operating system.[6] On top of this bottom layer is (1) a directory service that is used by the assembly controller to connect components and (2) a variety of other services used by components and assembly controllers.

This report does not document the programming interfaces to the Pin V1.0 runtime environment; as discussed in the next section, that part of the Pin component technology is being extensively reworked.

5. For more information, go to http://www.vci.com/embedded/products.aspx?ID=70.

6. When referring to Windows NT, we actually mean the Windows NT, Windows 2000, and Windows XP family of operating systems based on the Win32 API.

3 Current Work

The version of Pin described in this report has a significant dependency on a commercial software[7] that supports the development of real-time applications on Microsoft Windows NT. While this product was adequate for our immediate purpose and consistent with the assumptions of the λ_{ABA} performance theory [Hissam 03], we could not develop performance reasoning frameworks that exploit alternative scheduling disciplines, such as earliest deadline first (EDF). Beyond this restriction, the dependency on a commercial product limits our options for distributing Pin to clients or as an open source product.

For these (and other) reasons, we are currently rehosting Pin to our own virtual operating systems layer, called the Pin Kernel Services. This layer is currently implemented on Windows NT and Windows CE, but it can, in principle, be rehosted to any Unix variant[8] as well as to "bare" hardware. The major elements of the Pin kernel are shown in Figure 6.

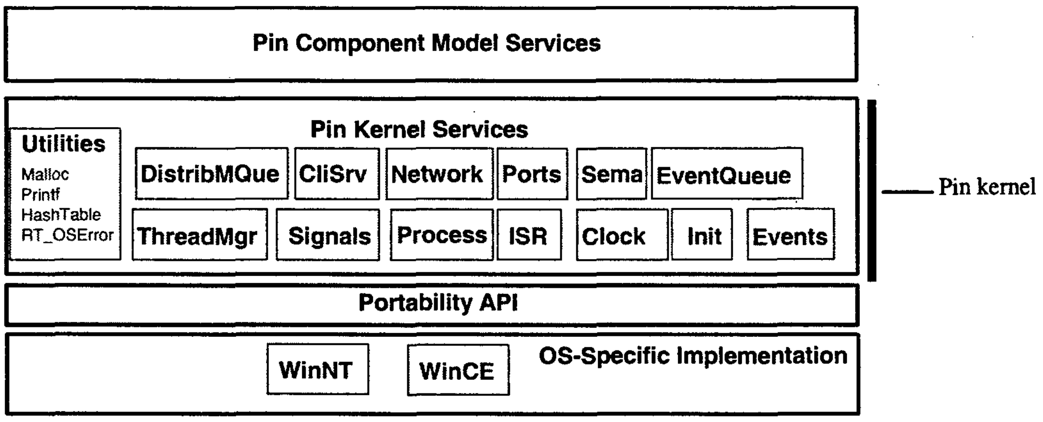


Figure 6: Pin Runtime Environment and the Pin Kernel

Revisions are being made to the Pin component model that will influence the structure of the Pin Component Model Services, shown in undifferentiated form in Figure 6. The two most significant areas of revision are in the treatment of event semantics and the support for distributed assemblies.

7. For more information, go to http://www.vci.com/embedded/products.aspx?ID=70.

8. In fact, the code base for our virtual operating system was initially developed as a prototype of POSIX real-time threads.

Pin event semantics are being revised to more directly reflect UML statechart semantics. In particular, a range of UML event classes will be supported by Pin, including (in addition to "pin" events) UML change and time events. It is important to note that the UML standard deliberately leaves semantic aspects of statecharts undefined, allowing some implementation latitude for UML tool vendors. We have selected a consistent semantics for statecharts within the space of allowed variation.

Pin V1.0 does not support distributed assemblies. Although we could (and did) handcraft solutions to permit components in one assembly, executing on one CPU, to communicate with components in another assembly on another CPU, these solutions were *ad hoc* and not reflected in the semantics of CCL. We are currently investigating several alternative approaches and will report on our progress in a future technical note.

4 Summary

Pin is a basic, simple component technology suitable for building embedded software applications. Pin implements the container idiom for software components. Containers provide a prefabricated "shell" in which custom code executes and through which all interactions between custom code and its external environment are mediated. Pin is a component technology for pure assembly (systems are assembled by selecting components and connecting their interfaces, which are composed of communication channels called pins) and has been used as a foundation for our work on PECTs.

This report describes the main concepts of Pin and documents the C-language interface to Pin V1.0. It also provides insight into some of the changes we are making currently to improve Pin. Table 1 summarizes those changes.

Table 1: Summary of Planned Improvements to Pin

Features	**Pin V1.0**	**Future Versions of Pin**
Supported operating systems	Windows NT	Windows NT, Windows CE, and potentially Unix variants
Real-time support	Provided by commercial product (RTX)	Provided by Pin kernel
Extensibility	None provided	Pluggable schedulers and communication mechanisms
Distribution	Assemblies restricted to a single processor	Support for distributed assemblies
Life cycle	The assembly is the controller, requiring runtime startup and shutdown for each assembly execution.	Provide a separate controller, allowing dynamic loading and unloading of assemblies across network boundaries
Component stimulus	Events representing communications among components are the only form of stimulus.	Additional event types introduced reflecting UML semantics for time and change events
Measurement support	Support for single processor measurement.	Will provide a distributed measurement infrastructure

Appendix A Pin's C API

A.1 Data Structures

The following are commonly used data types and structures supporting many of the component-supplied functions and Pin runtime support functions. Developers of assembly controllers will also refer to these functions.

IPC_MSG

Synopsis

```
#include <ComponentSpec.h>
#define  IPC_MAXMSGSIZE    1536

typedef struct
{
  IPC_HEADER header;
  char data[IPC_MAXMSGSIZE - sizeof (IPC_HEADER)];
} IPC_MSG;
```

Description

`IPC_MSG` is the structure used in all component interactions to send and receive messages, where

`header`	(reserved) used by the Pin runtime to dispatch messages. Elements within this portion of the structure should be considered opaque.
`data`	buffer used to store the message being dispatched

See Also

SendOutSourcePin(), SendOutSourcePinWait(), SendReply()

Example

```
IPC_MSG MessageOut;
SPrintf(MessageOut.data,"Clock From %s Pin %d",
  Reaction->Instance->UniqueName,0);
SendOutSourcePin (Reaction, 0, &MessageOut,
  (short)(Rt_strlen(MessageOut.data)+1),
  IPC_WAITFOREVER);
```

TCommonAnswer

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  IPC_MSG *MessageReply;
} TCommonAnswer;
```

Description

`TCommonAnswer` is the structure used to exchange data with the callback associated with a synchronous reply on a synchronous pin, as with `SendOutSourcePinWait()`. The callback can obtain the reply from within this data structure, where

`MessageReply`	message reply from the synchronous pin interaction

See Also

`IPC_MSG`, `SendOutSourcePinWait()`, `SendReply()`, `TCommonHandler()`

Example

```
TCommonAnswer *answer=(TCommonAnswer*)Data;
RtPrintf("Instance %s Pin %d Received %s\n",
   Reaction->Instance->UniqueName, 2,
   answer->MessageReply->data);
```

TCommonMsg

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  IPC_MSG *MessageIn;
  short    BytesIn;
  short    src_id;
  short    cmd;
  short    msg_type;
  short    user_def;
} TCommonMsg;
```

Description

TCommonMsg is the structure used to exchange data with common handlers when passing PINMSG (for instance, a reaction handler which is of type TCommonHandler() and TReason is PINMSG), where

MessageIn	message being set for a given pin interaction
BytesIn	size in bytes of the data portion of MessageIn
src_id	source pin from which the message was sent.
cmd	sink pin upon which the message was received
msg_type	user-defined field
user_def	user-defined field

See Also

IPC_MSG, TCommonHandler(), SendOutSourcePin(), SendOutSourcePinWait().

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
...
 if (Reason==PINMSG)
 {
  TCommonMsg *msg=(TCommonMsg *)Data;
  RtPrintf("R1 %s  Pin %d Received Message: %s\n",
   Reaction->Instance->UniqueName,
   msg->cmd, msg->MessageIn->data);
 }
}
```

TCommonTimeOut

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  long LastTimeout;
  long NextTimeout;
} TCommonTimeOut;
```

Description

`TCommonTimeOut` is the structure used to exchange timeout information with common handlers when passing `TIMEOUT` (for instance, a reaction handler which is of type `TCommonHandler()` and `TReason` is `TIMEOUT`), where

`LastTimeout`	last timeout value (in milliseconds) that expired, thereby causing the invocation of the reaction handler
`NextTimeout`	new timeout value (in milliseconds) to use upon return from the reaction for the next timeout. Setting `NextTimeout` to zero (0) will disable all future timeouts.

See Also

`TCommonHandler()`

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
...
 if (Reason==TIMEOUT)
 {
  TCommonTimeOut *to=(TCommonTimeOut *)Data;
  RtPrintf ("Reaction timeout at %ld\n", to->LastTimeout);
  to->NextTimeout = to->LastTimeout + 100; // new next timeout
 }
}
```

TCommonTmrEvent

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  REACTION_TIMER_HANDLE ReationTimerHandle;
  DWORD                 dwUser;
  DWORD                 NextDelay;
} TCommonTmrEvent;
```

Description

`TCommonTmrEvent` is the structure used to exchange timer information with common handlers when passing TMR (for instance, a reaction handler which is of type `TCommonHandler()` and `TReason` is TMR), where

`ReactionTimerHandle`	the system handle to the common handler just invoked. This handle is useful in destroying the handler associated with a timer.
`dwUser`	user-defined field
`NextDelay`	new timer value (in milliseconds) to use upon return from the reaction for the next timer. Setting `NextDelay` to zero (`0`) will cause the timer to expire immediately and `TCommonHandler()` to be invoked. Setting `NextDelay` to `INFINITE` will prevent the timer from expiring under any conditions.

See Also

`TCommonHandler(), CreateReactionTimer()`

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
...
 if (Reason==TMR)
 {
  TCommonTmrEvent *tmrevt=(TCommonTmrEvent *)Data;
  Time+=500;
  RtPrintf("ReactionHandler Instance %s Reaction %d\n",
   Reaction->Instance->UniqueName, tmrevt->dwUser);
 tmrevt->NextDelay=Time
 }
}
```

TComponentInfo

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  char          *Name;
  unsigned int  NumSourcePins;
  unsigned int  NumSinkPins;
  unsigned int  NumReactions;
} TComponentInfo;
```

Description

`TComponentInfo` consists of general, structural information about a component to include its name, number of source pins, and number of sink pins, and the total number of reactions it supports, where

`Name`	character string constant that is null terminated and identifies the name of the component
`NumSourcePins`	non-negative integer value indicating the total number of source pins
`NumSinkPins`	non-negative integer value indicating the total number of sink pins
`NumReactions`	non-negative integer value indicating the total number of reactions

Example

```
TComponentInfo ComponentInfo={
  "DISTCLOCK105",
  NUM_SOURCE_PINS,
  NUM_SINK_PINS,
  NUM_REACTIONS,
};
```

See Also

`GetNumSourcePins()`, `GetNumSinkPins()`, `GetNumReactions()`, `GetSourcePinInfo()`, `GetSinkPinInfo()`, `GetReactionInfo()`

TComponentInstance

Synopsis

```
#include <ComponentSpec.h>
typedef struct _TComponentInstance
{
  struct _TPinComponent    *Component;
  char                     *UniqueName;
  TSourcePins              *SourcePins;
  TReactions               *Reactions;
  void                     *InstanceData;
  struct _TReactionTimer
   *TimerHandlesIndexToPtr[MAX_REACTION_TIMER_HANDLES];
  int
   TimerHandlesFreeListNext[MAX_REACTION_TIMER_HANDLES];
  int                       TimerHandlesFreeList;
  CRITICAL_SECTION          TimerHandlesCriticalSection;
} TComponentInstance;
```

Description

`TComponentInstance` consists of general, structural information about an instance of a component to include the instance's name and data specific to a single instance, where

`Component`	handle to TPinComponent
`Name`	character string constant that is null terminated and identifies the unique name of the component instance
`SourcePins`	handle to the list of source pins
`Reactions`	handle to the list of source pins
`InstanceData`	handle to the list of source pins
`TimerHandlesIndexToPtr`	reserved
`TimerHandlesFreeListNext`	reserved

`TimerHandlesFreeList`	reserved
`TimerHandlesCriticalSection`	reserved

See Also

`ConfigureInstance(), DeleteInstance(), CreateInstance(), SourceAddSinkPin(), StartInstance(), StopInstance(), LoadComponent()`

Example

```
TComponentInstance *InstanceSimpleClock1;
if ((InstanceSimpleClock1=CreateInstance
  (PinSimpleClock, "clock1",
   &SimpleClockComponentProperties[0],
    sizeof(COMPONENT_SimpleClock_ARGS)))!=NULL)
{
  RtPrintf("Instance clock1 Created\n");
}
else
  RtPrintf("Failed to Create Instance clock1\n");
```

TPinComponent

Synopsis

```
#include <ComponentSpec.h>
typedef struct _TPinComponent
{
  HINSTANCE                     hLibModule;
  T_ABB_IPC_Functions           ABB_IPC_Functions;
  TPinDirServ                   PinDirServ;
  TContoller                    Controller;
  T_GetNumSinkPins              _GetNumSinkPins;
  T_GetNumSourcePins            _GetNumSourcePins;
  T_GetNumReactions             _GetNumReactions;
  T_GetReactionInfo             _GetReactionInfo;
  T_GetSourcePinInfo            _GetSourcePinInfo;
  T_GetSinkPinInfo              _GetSinkPinInfo;
  T_CreateInstance              _CreateInstance;
  T_DeleteInstance              _DeleteInstance;
  T_SourceAddSinkPin            _SourceAddSinkPin;
  T_SetReactionPriority         _SetReactionPriority;
  T_ConfigureInstance           _ConfigureInstance;
  T_SetReactionQueueLength      _SetReactionQueueLength;
  T_SetReactionTimeOut          _SetReactionTimeOut;
  T_StartInstance               _StartInstance;
  T_StopInstance                _StopInstance;
  T_SetMeasureExecutionTime     _SetMeasureExecutionTime;
} TPinComponent;
```

Description

`TPinComponent` is used as a system utility structure whose structure elements are all reserved.

See Also

`TComponentInstance`, `ConfigureInstance()`, `DeleteInstance()`, `CreateInstance()`, `SourceAddSinkPin()`, `StartInstance()`, `StopInstance()`, `LoadComponent()`

Example

```
TPinComponent *PinSimpleClock;
if ((PinSimpleClock=LoadComponent("SimpleClock.dll"))==NULL)
{
  RtPrintf("Failed to Load SimpleClock\n");
  ExitProcess(0);
}
else
  RtPrintf("SimpleClock Load Successful\n");
```

TPinInfoSink

Synopsis

```
#include <ComponentSpec.h>
typedef struct{
  char    *PinName;
  char    *PinType;
} TPinInfoSink;
```

Description

`TPinInfoSink` holds user-defined values for the name of the sink pin and the name of its type, where

`PinName`	character string constant that is null terminated and identifies the name of the sink pin
`PinType`	character string constant that is null terminated and identifies the type of the sink pin

The semantic significance of the values is user defined.

See Also

`SourceAddSinkPin()`

Example

```
TPinInfoSink  SinkPins[NUM_SINK_PINS]=
{
  {"Sporadic Server Request", "SS.request" }
};
```

TPinInfoSource

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  char    *PinName;
  char    *PinType;
} TPinInfoSource;
```

Description

`TPinInfoSource` holds user-defined values for the name of the source pin and the name of its type, where

`PinName`	character string constant that is null terminated and identifies the name of the source pin
`PinType`	character string constant that is null terminated and identifies the type of the source pin

The semantic significance of the values is user defined.

See Also

`SourceAddSinkPin()`

Example

```
TPinInfoSource  SourcePins[NUM_SOURCE_PINS]=
{
  {"Source Pin 0 (r0)", "TEST"},
  {"Source Pin 1 (r1)", "TEST"},
};
```

TReactions

Synopsis

```
#include <ComponentSpec.h>
typedef struct _TReactions
{
  unsigned int                 ReactionIndex;
  struct _TComponentInstance *Instance;
  short                        QueueSize;
  int                          Priority;
  TThreadInfo                  ThreadInfo;
  short                        IPC_SlotID;
  BOOL                         Valid;
  long                         TimeOut;
  BOOL                         MeasureExecutionTime;
} TReactions;
```

Description

The `TReactions` structure holds runtime information about an individual reaction and is passed as an argument to `TCommonHandler()`, where

`ReactionIndex`	non-negative integer representing the index of the reaction in the `TReactionsInfo` structure
`Instance`	character string constant that is null terminated and identifies the name of the reaction's component instance
`QueueSize`	non-negative integer of the queue size for holding messages for the reaction
`Priority`	non-negative integer denoting the reaction's priority
`ThreadInfo`	reserved
`IPC_SlotID`	reserved

`Valid`	reserved
`TimeOut`	non-negative integer indicating the timeout for the reaction that will generate a `TIMEOUT` message should no other message type arrive
`MeasureExecutionTime`	Boolean indicating whether (`TRUE`) or not (`FALSE`) measurement events should be generated by the reaction

See Also

`TCommonHandler(), TReactionInfo`

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
...
 if (Reason==PINMSG)
 {
  TCommonMsg *msg=(TCommonMsg *)Data;
  RtPrintf("R1 %s  Pin %d Received Message: %s\n",
   Reaction->Instance->UniqueName,
   msg->cmd, msg->MessageIn->data);
 }
}
```

TReactionsInfo

Synopsis

```
#include <ComponentSpec.h>
typedef struct
{
  unsigned int    NumSourcePins;
  unsigned int    NumberOfSinkPins;
  unsigned int    *SourcePins;
  unsigned int    *SinkPins;
  short           DefaultQueueSize;
  int             DefaultPriority;
  long            DefaultTimeOut;
  BOOL            DefaultMeasureExecutionTime;
  TCommonHandler  Handler;
  TCommonHandler  TimeoutHandler;
} TReactionsInfo;
```

Description

The `TReactionsInfo` structure is initialized at compile time and accessed by the system to acquire information about the handlers in the system and the associated source and sink pins related to that handler, where

`NumSourcePins`	non-negative integer indicating the number of source pins associated with the handler
`NumberOfSinkPins`	non-negative integer indicating the number of sink pins associated with the handler
`SourcePins`	list of source pins (whose number matches Num-SourcePins) associated with the handler
`SinkPins`	list of sink pins (whose number matches `Num-berOfSinkPins`) associated with the handler
`DefaultQueueSize`	non-zero integer indicating the size of the input queue to the handler for inbound message inter-actions

`DefaultPriority`	non-negative integer for the default priority for the handler, used for scheduling
`DefaultTimeOut`	non-negative integer for the default timeout to use before invoking the handler after no messages have been received. A value of 0 disables timeouts.
`DefaultMeasureExecutionTime`	Boolean value of whether (TRUE) or not (FALSE) measurement traces should be emitted for pin interactions with this handler
`Handler`	pointer to the normal message handler
`TimeoutHandler`	pointer to the timeout handler (which can be the same as the normal handler)

See Also

`GetReactionInfo()`

Example

```
TReactionsInfo   ReactionInfo[NUM_REACTIONS]=
{
   {
    REACTIONS_0_NUM_SINKS,
    REACTIONS_0_NUM_SOURCE,
    NULL,
    Reactions_0_Source,
    5,
    10,
    IPC_WAITFOREVER,
    FALSE,
    Reaction_0_TimeoutHandler
   }
};
```

TReason

Synopsis

```
#include <ComponentSpec.h>
typedef enum
{
  PINMSG,
  TMR,
  TIMEOUT,
  ANSWER
} TReason;
```

Description

`TReason` is used to differentiate between messages passed to `TCommonHandler()`, where

`PINMSG`	indicates a message conforming to `TCommonMsg`
`TMR`	indicates a message conforming to `TCommonTmrEvent`
`TIMEOUT`	indicates a message conforming to `TCommonTimeOut`
`ANSWER`	indicates a message conforming to `TCommonAnswer`

See Also

`TCommonHandler(), SendReply(), CreateReactionTimer(), TReactionsInfo`

Example

```
BOOL Reaction_0_Handler
  (TReactions *Reaction, TReason Reason, void * Data)
{
 if (Reason==PINMSG)
 {
   // Handle TCommonMsg
 }
 else if (Reason==TMR)
 {
   // Handle TCommonTmrEvent
 }
 else if (Reason==TIMEOUT)
 {
   // Handle TCommonTimeOut
 }
 else if (Reason==ANSWER)
 {
   // Handle TCommonAnswer
 }
}
```

A.2 User Code API

Functions appearing in this section are functions that are required to be provided by the user-supplied portion of the component. The container will invoke these functions as per the Pin component life cycle.

CreateComponentInstance

Synopsis

```
#include <ComponentFuncs.h>
BOOL CreateComponentInstance(
   void      **Data,
   void      *State,
   unsigned int SizeOfState
);
```

Description

`CreateComponentInstance()` is called when an instance of a component is to be created. It is the first instance-specific call made by the Pin runtime. This function should create all instance-specific data and state information relevant to an instance object. The data and state information created are intended to be private to this instance. Any specific initialization data and state information relevant to this component and instance are passed in the `State` pointer. If this data and state are to persist for the lifetime of the instance created, heap memory allocation should be performed in this function, and references to that memory created can be stored as callback in `Data`.

Component defined data in `State` is not guaranteed to persist after the call to `CreateComponentInstance()` and must be saved prior to a return from this function.

`Data`	non-null pointer provided by the Pin runtime and used by an instance of a component to save instance-specific data and state information. The pointer set by `CreateComponentInstance()` is saved in the `TReaction` data structure element `TComponentInstance.InstanceData.`
`State`	If non-null, the value is a pointer provided by the controller to data and state information intended

	to be specific to the instance of a component that is being created. A NULL value indicates that no data or state information was passed.
SizeOfState	If State is non-null, SizeOfState is a positive non-zero integer indicating the number of bytes in memory of State.

Return Values

TRUE	indicates that CreateComponentInstance() was successfully able to create an instance of a component
FALSE	indicates that CreateComponentInstance() failed to successfully create an instance of a component

See Also

CreateInstance()

Example

```
BOOL CreateComponentInstance(
   void **Data, void *State, unsigned int SizeOfState)
{
 if (Data==NULL) return(FALSE);
 else *Data=NULL;

 if (State != NULL) {
   *Data =
      RtAllocateLockedMemory(SizeOfState);
   if (*Data == NULL)
      return (FALSE);
   Rt_memcpy (
      *Data, State, sizeof(SizeOfState));
 }
 return (TRUE);
}
```

DeleteComponentInstance

Synopsis

```
#include <ComponentFuncs.h>
BOOL DeleteComponentInstance(
   void **Data
);
```

Description

`DeleteComponentInstance()` is called when an instance of a component is to be deleted, and there are no further references to a component instance to be made. This is the last instance-specific call made by the Pin runtime. This function should free all instance-specific data and state information relevant to an instance object from the heap.

`Data`	non-null pointer provided by the Pin runtime and used to access instance-specific data and state information

Return Values

`TRUE`	indicates that `DeleteComponentInstance()` was successfully able to destroy an instance of a component
`FALSE`	indicates that `DeleteComponentInstance()` failed to successfully destroy an instance of a component

See Also

`DeleteInstance()`

Example

```
BOOL DeleteComponentInstance(void **Data)
{
 if (Data==NULL) return(FALSE);
 else {
    if (*Data) RtFreeLockedMemory (*Data);
    *Data=NULL;
 }
 return(TRUE);
}
```

ReactionInitialize

Synopsis

```
#include <ComponentFuncs.h>


void ReactionInitialize (
   TReactions *Reaction,
   int ReactionIndex
);
```

Description

`ReactionInitialize()` is called after an instance of a component is created and before the component instance is to receive messages on sink pins. This function is called during the component instance's startup phase and is invoked by `StartInstance()`. Any reaction-specific actions that need to be completed before messages are received should be done here (e.g., creating timers).

`Reaction`	non-null pointer provided by the Pin runtime and used by an instance's reaction to access instance- and reaction-specific data and state information
`ReactionIndex`	non-negative integer value indicating the index of the reaction being initialized. This index is determined by the total number of reactions and the `TReactionsInfo` statically declared for the component.

Return Values

None

See Also

`StartInstance()`

Example

```
void ReactionInitialize
   (TReactions *Reaction,int ReactionIndex)

TUserData *Data;
Data=(TUserData *) Reaction->Instance->InstanceData;

if ((Data->Handle[ReactionIndex]=
   CreateReactionTimer
     (Reaction,ReactionTmrHandler,12,500,
      TRUE,ReactionIndex))== NULL_INDEX )
{
   RtPrintf("Initialize Reaction Error\n");
}
else
   RtPrintf("Initialize Reaction OK\n");
}
```

ReactionTerminating

Synopsis

```
#include <ComponentFuncs.h>
void ReactionTerminating (
   TReactions *Reaction,
   int ReactionIndex
);
```

Description

`ReactionTerminating()` is called prior to terminating a component instance, indicating that reactions handling messages on the sink pin should cease. This function is called during the component instance's shutdown phase and is invoked by `StopInstance()`. Any reaction-specific actions that need to be completed before terminating the component instance should be done here (e.g., destroying timers).

`Reaction`	non-null pointer provided by the Pin runtime and used by an instance's reaction to access instance- and reaction-specific data and state information
`ReactionIndex`	non-negative integer value indicating the index of the reaction being terminated. This index is determined by the total number of reactions and the `TReactionsInfo` statically declared for the component.

Return Values

None

See Also

`StopInstance()`

Example

```
void ReactionTerminating
   (TReactions *Reaction,int ReactionIndex)
{
TUserData *Data;
Data=(TUserData *) Reaction->Instance->InstanceData;

if (DestroyReactionTimer
   (Reaction,Data->Handle[ReactionIndex])==TRUE)
   RtPrintf("Terminating Reaction OK\n");
else
   RtPrintf("Terminating Reaction Failed\n");
}
```

TCommonHandler

Synopsis

```
#include <ComponentSpec.h>
typedef BOOL (*TCommonHandler) (
   struct _TReactions *Reaction,
   TReason Reason,
   void * Data
);
```

Description

`TCommonHandler()` is a function prototype and not a specific function to be provided by the component developer. The Pin runtime will invoke specified handlers according to the data structures created by the component developer, such as `TReactionsInfo.Handler` and `TReactionsInfo.TimeoutHandler`, or provided as parameters to other functions (e.g., `CreateReactionTimer()`). Such handlers must conform to the type definition for this function prototype.

Each handler is called with three things: (1) a pointer to a structure containing information about the reaction, (2) the reason the handler was invoked, and (3) callback data and state information for the specific instance of the component to which this handler belongs.

`Reaction`	non-null pointer provided by the Pin runtime and used by an instance's reaction to access instance- and reaction-specific data and state information
`Reason`	reason why `TCommonHandler()` was called: `PINMSG`, `TIMEOUT`, `TMR`, or `ANSWER`
`Data`	a non-null pointer provided by the Pin runtime and used to access instance-specific data and state information

Return Values

`TRUE`	indicates to the Pin runtime that the handler encountered no reportable errors

`FALSE`	indicates to the Pin runtime that the handler encountered an exception

See Also

`TReactionInfo, TReason, SendOutSourcePin(), SendOutSourcePin-Wait(), SendReply(), CreateReactionTimer()`

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
 if (Reason==PINMSG)
 {
  RtPrintf("R1 %s  Pin %d Received Message: %s\n",
   Reaction->Instance->UniqueName,
   msg->cmd, msg->MessageIn->data);
 }
 else if (Reason==TMR)
 {
  RtPrintf("R1 Timer Expired.\n");
 }
}
```

A.3 Container API

User-supplied component code is provided for the functions in this section to manage behavior and interactions with other components and the Pin runtime.

CreateReactionTimer

Synopsis

```
#include <ComponentSpec.h>
REACTION_TIMER_HANDLE CreateReactionTimer (
   TReactions *Reaction,
   TCommonHandler Callback,
   int Priority,
   DWORD Delay,
   BOOL Periodic,
   DWORD dwUser
);
```

Description

`CreateReactionTimer()` will create a timer having a specified delay and priority that will invoke a specific timer handler (i.e., callback). When a timer expires, a TMR message is placed on the reaction queue for the specified timer handler.

`Reaction`	non-null pointer indicating which instance's reaction the timer should be assigned to
`Callback`	non-null pointer indicating the `TCommonHandler()` handler to call when the timer expires
`Priority`	non-negative integer indicating the priority at which the timer is to operate. The range is `0 >= Priority <= 255`.
`Delay`	non-negative integer indicating the timer's delay before expiring. A zero delay will cause the timer to expire immediately (while at its assigned priority).

`Periodic`	If TRUE, the timer will continue to expire at the given priority and rate (delay). If FALSE, the timer will expire only once.
`dwUser`	pointer to user-defined callback data. This pointer will be provided to the callback when the timer expires.

Return Values

A failure will result in a return value of NULL_INDEX; otherwise the return value is a valid REACTION_TIMER_HANDLE.

Errors

A reaction can have at most MAX_REACTION_TIMER_HANDLES timer handles. `CreateReactionTimer()` will fail with NULL_INDEX if more than MAX_REACTION_TIMER_HANDLES timer handlers are being created. Use `DestroyReactionTimer()` to free up unused handles or handles that are no longer needed.

See Also

`DestroyReactionTimer()`

Example

```
void ReactionInitialize
   (TReactions *Reaction,int ReactionIndex)
{
 TUserData *Data;
 Data=(TUserData *) Reaction->Instance->InstanceData;

 if ((Data->Handle[ReactionIndex]=
   CreateReactionTimer
     (Reaction,ReactionTmrHandler,12,500,
      TRUE,ReactionIndex))== NULL_INDEX )
 {
   RtPrintf("Initialize Reaction Error\n");
 }
 else
   RtPrintf("Initialize Reaction OK\n");
}
```

DestroyReactionTimer

Synopsis

```
#include <ComponentSpec.h>
BOOL DestroyReactionTimer(
   TReactions *Reaction,
   REACTION_TIMER_HANDLE Handle
);
```

Description

`DestroyReactionTimer()` will destroy or otherwise remove a timer created with `CreateReactionTimer()`. The timer destroyed will no longer generate messages for the specified reaction.

`Reaction`	non-null pointer indicating from which instance's reaction the timer should be destroyed
`Handle`	handle to the specific timer to be destroyed. This handle was previously returned by `CreateReactionTimer()`.

Return Values

`TRUE` if the timer was destroyed successfully, `FALSE` otherwise.

Errors

`Handle` must have been returned by `CreateReactionTimer()` and is in the range of `0 < Handle <= MAX_REACTION_TIMER_HANDLES`.

See Also

`CreateReactionTimer()`

Example

```
void ReactionTerminating
   (TReactions *Reaction,int ReactionIndex)
{
 TUserData *Data;
 Data=(TUserData *) Reaction->Instance->InstanceData;

 if (DestroyReactionTimer
   (Reaction,Data->Handle[ReactionIndex])==TRUE)
   RtPrintf("Terminating Reaction OK\n");
 else
   RtPrintf("Terminating Reaction Failed\n");
}
```

`NotifyController`

Synopsis

```
#include <ComponentSpec.h>
int NotifyContoller (
   TComponentInstance *Instance,
   int code,
   char *string
);
```

Description

`NotifyController()` is used by component instances to send the following to the controller of another instance: (1) a user-defined message or (2) code and an optional character string. This function is useful in communicating exception conditions to the controller, requiring controller attention (such as shutdown).

`Instance`	non-null pointer indicating which component instance is performing the notification
`code`	user-defined integer (optional) that is passed to the controller and is application dependent
`string`	user-defined string (optional) that is passed to the controller and is application dependent

Return Values

SUCCESS	non-null pointer indicating which component instance is performing the notification
CONTROLLER_NOT_FOUND	Either the shared memory segment for message passing has not been initialized (via `StartPinInterface()`) or the queue for delivering the message to the controller has been deleted.
INVALID_INSTANCE	An instance was passed as a null pointer.
INVALID_STRING	A string was passed as a null pointer.

INVALID_INSTANCE_UNIQUE_NAME	The instance name was either of zero length (an empty, but non-null string) or too long.
STRING_MESSAGE_TOO_LONG	The string was either of zero length (an empty, but non-null string) or too long.
CONTROLLER_QUEUE_FULL	The queue for delivering message to the controller is full.
CONTROLLER_UNKNOWN_ERROR	An error of the underlying IPC mechanism for delivering messages has occurred for an unknown reason.

Errors

Errors generated by this function and the conditions are described above under "Description."

See Also

WaitForNotifications()

Example

```
BOOL Reaction_1_Handler
   (TReactions *Reaction, TReason Reason, void * Data)
{
 static DWORD Time=0;
 if (Reason==PINMSG)
 {
  if (NotifyContoller(Reaction->Instance,
      543212345,"It works")==SUCCESS)
    RtPrintf("Done Notification Sent\n");
  else
    RtPrintf("Notification Send Error\n");
 }
}
```

SendOutSourcePin

Synopsis

```
#include <ComponentSpec.h>
BOOL SendOutSourcePin (
   TReactions *Reaction,
   unsigned int SourcePin,
   IPC_MSG *MessageOut,
   short MsgSize,
   long Timeout
);
```

Description

`SendOutSourcePin()` is used to send out an asynchronous message via a component instance's source (or stimulus) pin. `SendOutSourcePin()` will block long enough to queue the message on the one or more queues of the connected, and interoperating, sink (or receiving) pins.

`Reaction`	non-null pointer indicating the reaction generating the message
`SourcePin`	index of the source pin on which the message is being generated
`MessageOut`	message being sent
`MsgSize`	size in bytes of `MessageOut`
`Timeout`	the length of time in milliseconds to wait for delivering messages that would block (e.g., in the case of a full queue). A timeout of IPC_WAITFOREVER will cause `SendOutSourcePin()` to wait until the message can be delivered to the queue of the destination sink pin.

Return Values

TRUE	`SendOutSourcePin()` was able to queue the message for delivery on the destination sink pin(s).
FALSE	`SendOutSourcePin()` failed to queue the message for delivery on the destination sink pin(s).

Errors

`SendOutSourcePinWait()` will fail if the component instance does not have any source pins or failed in the IPC code because of a timeout, message size error, invalid source pin ID, or unrecoverable system error.

See Also

`SendOutSourcePinWait()`

Example

```
BOOL Reaction_0_TimeoutHandler
  (TReactions *Reaction, TReason Reason, void * Data)
{
 if (Reason==TIMEOUT)
 {
  IPC_MSG MessageOut;
  TCommonTimeOut *to=(TCommonTimeOut *) Data;
  SPrintf(MessageOut.data,"Clock From %s Pin %d",
    Reaction->Instance->UniqueName,0);
  if (!SendOutSourcePin
    (Reaction,0,&MessageOut,
    (short)(Rt_strlen(MessageOut.data)+1),
    IPC_WAITFOREVER))
      RtPrintf("SinkPin 1 Handler Send Error\n");
  RtPrintf("Clock %s Sent Trigger \n",
    Reaction->Instance->UniqueName);
  to->NextTimeout=to->LastTimeout;
 }
 return(TRUE);
}
```

SendOutSourcePinWait

Synopsis

```
#include <ComponentSpec.h>
BOOL SendOutSourcePinWait (
   TReactions *Reaction,
   unsigned int SourcePin,
   IPC_MSG *MessageOut,
   short MsgOutSize,
   IPC_MSG *MessageIn,
   short MsgInSize,
   long Timeout,
   TCommonHandler Callback
);
```

Description

`SendOutSourcePinWait()` is used to send a synchronous message out via a component instance's source (or stimulus) pin. `SendOutSourcePinWait()` will block until the reaction of the connected component instance's sink (or receiving) pin has issued a `SendReply()`. `SendOutSourcePinWait()` provides a callback mechanism to the connected sink pin's reaction to reply to a synchronous message.

`Reaction`	non-null pointer indicating the reaction generating the message
`SourcePin`	index of the source pin on which the message is being generated
`MessageOut`	message being sent
`MsgOutSize`	size in bytes of `MessageOut`
`MessageIn`	buffer to hold the response message
`MsgInSize`	size in bytes of the buffer to hold the response message
`Timeout`	length of time in milliseconds to wait for delivering messages that would block (e.g., in the case

	of a full queue). A timeout of `IPC_WAITFOREVER` will cause `SendOut-SourcePinWait()` to wait until the message can be delivered to the queue of the destination sink pin.
`Callback`	(optional) If non-null, the value is the `TCom-monHandler()` to use as a callback for handling the messages returned in `MessageIn`.

Return Values

`TRUE`	indicates that `SendOutSourcePinWait()` was able to queue the message for delivery and receive acknowledgement of the message's receipt on the destination sink pin
`FALSE`	indicates that `SendOutSourcePinWait()` failed to queue the message for delivery or receive acknowledgement of the message's receipt on the destination sink pin

Errors

`SendOutSourcePinWait()` will fail if the component instance does not have any source pins or failed in the IPC code because of a timeout, message size error, invalid source pin ID, or unrecoverable system error.

See Also

`SendReply(), SendOutSourcePin()`

Example

```
BOOL Reaction_1_Handler
  (TReactions *Reaction,TReason Reason, void * Data)
{
 static DWORD Time=0;
 if (Reason==PINMSG)
 {
  IPC_MSG MessageOut;
  IPC_MSG MessageReplyRecv;
  TCommonMsg *msg=(TCommonMsg *) Data;
  SPrintf(MessageOut.data,
    "From %s Pin %d",Reaction->Instance->UniqueName,2);
  if (!SendOutSourcePinWait(Reaction, 2,
    &MessageOut,(short)(Rt_strlen(MessageOut.data)+1),
    &MessageReplyRecv,
    IPC_MAXMSGSIZE - sizeof(IPC_HEADER),
    IPC_WAITFOREVER, SinkPin_2_AnswerCallback))
      RtPrintf("SinkPin 2 Handler Send Error\n");
  RtPrintf("H2 %s  Pin %d Received Message: %s\n",
    Reaction->Instance->UniqueName,
    msg->cmd,msg->MessageIn->data);
 }
}
```

SendReply

Synopsis

```
#include <ComponentSpec.h>
BOOL SendReply (
   TReactions *Reaction,
   short src_id,
   IPC_MSG *MessageReply,
   short MsgReplySize
);
```

Description

`SendReply()` is used to send a reply to a received synchronous message via `SendOutSourcePinWait()`. The reaction that initiated the synchronous message will not unblock until a `SendReply()` is initiated from the reaction handling the sink pin to which the synchronous message was sent.

`Reaction`	non-null pointer indicating the reaction generating the reply message
`src_id`	index of the sink pin from which the reply message is being generated
`MessageReply`	reply message being sent
`MsgReplySize`	size in bytes of `MessageReply`

Return Values

`TRUE`	indicates that `SendReply()` was able to queue the reply message for delivery on the source pin that initiated the message
`FALSE`	indicates that `SendReply()` failed to queue the reply message for delivery on the source pin that initiated the message

Errors

SendReply() will fail if the component instance does not have any sink pins or failed in the IPC code because of a message size error, invalid sink pin id, or unrecoverable system error.

See Also

SendOutSourcePinWait()

Example

```
BOOL Reaction_1_Handler
  (TReactions *Reaction,TReason Reason, void * Data)
{
 static DWORD Time=0;
 if (Reason==PINMSG)
 {
  IPC_MSG MessageOut;
  IPC_MSG MessageReplyRecv;
  IPC_MSG MessageReply;
  TCommonMsg *msg=(TCommonMsg *) Data;
  SPrintf(MessageOut.data,
    "From %s Pin %d",Reaction->Instance->UniqueName,2);
  if (!SendOutSourcePinWait(Reaction, 2,
    &MessageOut,(short)(Rt_strlen(MessageOut.data)+1),
    &MessageReplyRecv,
    IPC_MAXMSGSIZE - sizeof(IPC_HEADER),
    IPC_WAITFOREVER, SinkPin_2_AnswerCallback))
      RtPrintf("SinkPin 2 Handler Send Error\n");
  RtPrintf("H2 %s  Pin %d Received Message: %s\n",
    Reaction->Instance->UniqueName,
    msg->cmd,msg->MessageIn->data);
  if (msg->MessageIn->header.msg_type== IPC_SENDWAIT)
  {
   SPrintf(MessageReply.data,
     "Reply From %s Pin %d",Reaction->Instance->UniqueName,2);
   SendReply(Reaction,msg->src_id,
     &MessageReply,(short)(Rt_strlen(MessageReply.data)+1));
  }
 }
}
```

A.4 Component API

These functions are used by assembly controllers to manage the life cycle of component instances to perform a function specific to the assembly.

`ConfigureInstance`

Synopsis

```
#include <PinInterface.h>
BOOL ConfigureInstance (
   TComponentInstance *Instance
);
```

Description

`ConfigureInstance()` is used to allow a newly created instance of a component to initialize and perform any necessary pre-startup configuration prior to `StartInstance()`.

`Instance`	non-null pointer indicating which component instance is to be configured

Return Values

TRUE	indicates that `ConfigureInstance()` was able to properly configure the component instance and its reactions and sink and source pins
FALSE	indicates that `ConfigureInstance()` failed to properly configure the component instance and its reactions and sink and source pins

Errors

The component instance may fail to properly initialize under a number of conditions, including failure to allocate necessary memory, an improperly specified number of sink or source pins, or connect to necessary IPC slots.

See Also

`CreateInstance(), LoadComponent(), SetReactionTimeOut(), SetReactionPriority(), SetMeasureExecutionTime(), StartPinInterface()`

Example

```
TComponentInstance *InstanceSimpleClock1;
...
if (ConfigureInstance(InstanceSimpleClock1)==FALSE)
{
  RtPrintf
    ("ConfigureInstance Instance Simple Clock1 FAILED \n");
}
```

CreateInstance

Synopsis

```
#include <PinInterface.h>


TComponentInstance* CreateInstance(
   struct _TPinComponent *Component,
   char *UniqueName,
   void *State,
   unsigned int SizeOfState
);
```

Description

`CreateInstance()` is used to create an instance of a loaded component library. The component instance must be created with a unique name and can pass initialization data or state information to the component as defined by the specific component specification. The passed initialization data or state information need not persist for the duration of the component instance's life cycle (it is the component instance's responsibility to save off this passed data).

`Component`	non-null pointer to heap or stack memory where `CreateInstance()` can store runtime-specific information about the newly created component instance
`UniqueName`	non-null, non-empty character string (null-terminated) that is unique among all the other component instances that have been or will be created
`State`	If non-null, this value is a pointer provided by the controller to data and state information that is intended to be specific to the instance of a component being created. If `NULL`, no data or state information was passed.
`SizeOfState`	If `State` is non-null, `SizeOfState` is a positive non-zero integer indicating the number of bytes in memory of `State`.

Return Values

`TComponentInstance`	If successful, `CreateInstance()` will return a non-null pointer to an instance of a component.
`NULL`	`CreateInstance()` will return a null pointer if it fails to create an instance of a component.

Errors

The component may fail to create an instance under a number of conditions, including failure to allocate necessary memory or improperly specified parameters (such as an invalid unique name).

See Also

`LoadComponent(), StartPinInterface()`

Example

```
TComponentInstance *InstanceSimpleClock1;
...
if ((InstanceSimpleClock1=CreateInstance
   (PinSimpleClock, "clock1",
    &SimpleClockComponentProperties[0],
     sizeof(COMPONENT_SimpleClock_ARGS)))!=NULL)
{
   RtPrintf("Instance clock1 Created\n");
}
else
   RtPrintf("Failed to Create Instance clock1\n");
```

DeleteInstance

Synopsis

```
#include <PinInterface.h>
BOOL DeleteInstance (
   TComponentInstance *Instance
);
```

Description

`DeleteInstance()` is used to destroy an instance of a component. Once a component instance is destroyed, no further references to that instance are valid.

`Instance`	non-null pointer returned by `CreateInstance()` of the component instance to be destroyed

Return Values

`TRUE`	`DeleteInstance()` was able to properly destroy the component instance and its reactions and sink and source pins.
`FALSE`	`DeleteInstance()` failed to properly destroy the component instance and its reactions and sink and source pins.

Errors

The component instance will fail only under one condition—when passing a `NULL` pointer.

See Also

`CreateInstance()`

Example

```
TComponentInstance *InstanceSimpleClock1;
...
if (InstanceSimpleClock1) DeleteInstance(InstanceSimpleClock1);
```

GetNumReactions

Synopsis

```
#include <PinInterface.h>
unsigned int GetNumReactions (TPinComponent *Component);
```

Description

`GetNumReactions()` is used to read or obtain the total number of reactions supported by a component.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`

Return Values

`int`	total number of reactions supported by the component

Errors

The passing of an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, will fail.

See Also

`LoadComponent()`

Example

```
TPinComponent *PinSimpleClock;
PinSimpleClock=LoadComponent("SimpleClock.dll");
RtPrintf("Component has %d reactions\n",
  GetNumReactions(PinSimpleClock));
```

`GetNumSinkPins`

Synopsis

```
#include <PinInterface.h>
unsigned int GetNumSinkPins (TPinComponent *Component);
```

Description

`GetNumSinkPins()` is used to read or obtain the total number of sink pins supported by a component.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`

Return Values

`int`	total number of sink pins supported by the component

Errors

The passing of an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, will fail.

See Also

`LoadComponent()`

Example

```
TPinComponent *PinSimpleClock;
PinSimpleClock=LoadComponent("SimpleClock.dll");
RtPrintf("Component has %d sink pins\n",
  GetNumSinkPins(PinSimpleClock));
```

GetNumSourcePins

Synopsis

```
#include <PinInterface.h>
unsigned int GetNumSourcePins (TPinComponent *Component);
```

Description

`GetNumSourcePins()` is used to read or obtain the total number of source pins supported by a component.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`

Return Values

`int`	total number of source pins supported by the component

Errors

The passing of an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, will fail.

See Also

`LoadComponent()`

Example

```
TPinComponent *PinSimpleClock;
PinSimpleClock=LoadComponent("SimpleClock.dll");
RtPrintf("Component has %d sink pins\n",
  GetNumSourcePins(PinSimpleClock));
```

GetReactionInfo

Synopsis

```
#include <PinInterface.h>
TReactionsInfo* GetReactionInfo (
   TPinComponent *Component;
   unsigned int Reaction
);
```

Description

`GetReactionInfo()` is used to read or obtain the information about a specific reaction handler for a component. Indices to reactions start at `0`.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`
`Reaction`	non-negative integer, which is the index into `TReactionsInfo` for the reaction information to be retrieved

Return Values

`TReactionsInfo`	`GetReactionInfo()` will return a non-null pointer to the specified reaction information for a component.

Errors

The function will fail if

- the reaction index is less than `0` or greater than or equal to the number of reactions supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed

See Also

`TReactionsInfo, CreateInstance()`

Example

```
TPinComponent *PinSimpleClock;
TReactionsInfo *r;
PinSimpleClock=LoadComponent("SimpleClock.dll");
r = GetReactionInfo(PinSimpleClock, 0);
RtPrintf("Default timeout for reaction 0 is %d\n",
  r->DefaultTimeOut);
```

GetSinkPinInfo

Synopsis

```
#include <PinInterface.h>
TPinInfoSink* GetSinkPinInfo (
   TPinComponent *Component;
   unsigned int Num
);
```

Description

`GetSinkPinInfo()` is used to read or obtain the information about a specific sink pin for a component. Indices to sink pins start at 0.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`
`Num`	non-negative integer, which is the index into `TPinInfoSink` for the sink pin information needed to retrieve

Return Values

`TPinInfoSink`	`GetSinkPinInfo()` will return a non-null pointer to the specified sink pin information for a component.

Errors

The function will fail if

- the sink pin index is less than 0 or greater than or equal to the number of sink pins supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed

See Also

`TPinInfoSink, CreateInstance()`

Example

```
TPinComponent *PinSimpleClock;
TPinInfoSink *pin;
PinSimpleClock=LoadComponent("SimpleClock.dll");
pin = GetSinkPinInfo(PinSimpleClock, 0);
RtPrintf("Name for sink pin 0 is %s\n",
  pin->PinName);
```

GetSourcePinInfo

Synopsis

```
#include <PinInterface.h>
TPinInfoSource* GetSourcePinInfo (
   TPinComponent *Component;
   unsigned int Num
);
```

Description

`GetSourcePinInfo()` is used to read or obtain the information about a specific source pin for a component. Indexes to source pins start at 0.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`
`Num`	non-negative integer, which is the index into `TPinInfoSource` for the source pin information to be retrieved

Return Values

`TPinInfoSource`	`GetSourcePinInfo()` will return a non-null pointer to the specified source pin information for a component.

Errors

The function will fail if

- the source pin index is less than 0 or greater than or equal to the number of source pins supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed

See Also

`TPinInfoSource, CreateInstance()`

Example

```
TPinComponent *PinSimpleClock;
TPinInfoSource *pin;
PinSimpleClock=LoadComponent("SimpleClock.dll");
pin = GetSourcePinInfo(PinSimpleClock, 0);
RtPrintf("Name for source pin 0 is %d\n",
  pin->PinName);
```

LoadComponent

Synopsis

```
#include <PinInterface.h>
TPinComponent LoadComponent (
   char *ComponentName
);
```

Description

LoadComponent() is used to dynamically load a component into the memory space of an assembly controller. LoadComponent() uses operating-system-specific libraries to dynamically load code modules (e.g., LoadLibrary() for the Win32 API) into the assembly's address space. The name is conformant to the naming convention of the deployment platform and conformant to the rules of the underlying native libraries.

ComponentName	non-null pointer, non-empty string of the name of the component to load into the assembly's address space

Return Values

TPinComponent	LoadComponent() will return a non-null pointer to the loaded pin component.
NULL	LoadComponent() will fail if it could not load the pin component.

Errors

Passing a null string pointer or an empty string will result in a failure. A failure can occur if the component specified in the parameter does not conform to the Pin component API. Also, a failure can occur if the component name specified cannot be found by the native libraries or the name does not conform to the conventions of those libraries.

See Also

UnloadComponent(), LoadLibrary() for the Win32 API.

Example

```
TPinComponent *PinSimpleClock;
if ((PinSimpleClock=LoadComponent("SimpleClock.dll"))==NULL){
  RtPrintf("Failed to Load SimpleClock\n");
  ExitProcess(0);
}
else
  RtPrintf("SimpleClock Load Successful\n");
```

SetMeasureExecutionTime

Synopsis

```
#include <PinInterface.h>
BOOL SetMeasureExecutionTime (
   TComponentInstance *Instance,
   unsigned int Reaction,
   BOOL Measure
);
```

Description

`SetMeasureExecutionTime()` is used to enable or disable the emission of measurement trace events by a component instance's reaction.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance of the reaction to be measured
`Reaction`	non-negative integer, which is the index into `TReactionsInfo` for the reaction for which the measurement flag is to be set
`Measure`	`TRUE` enables measurement of the component instance's reaction; `FALSE` disables it.

Return Values

`TRUE`	`SetMeasureExecutionTime()` was able to set the measurement flag for the reaction.
`FALSE`	`SetMeasureExecutionTime()` failed to set the measurement flag for the reaction.

Errors

The function will fail if

- the reaction index is less than `0` or greater than or equal to the number of reactions supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed

See Also

`CreateInstance()`

Example

```
if (!SetMeasureExecutionTime(InstanceSimpleClock1,0,TRUE))
{
  RtPrintf
   ("failed to set reaction 0 measurement for SimpleClock1\n");
}
```

`SetReactionPriority`

Synopsis

```
#include <PinInterface.h>
BOOL SetReactionPriority (
   TComponentInstance *Instance,
   unsigned int Reaction,
   int Priority
);
```

Description

`SetReactionPriority()` is used to set the priority of a component instance's reaction. The value set overwrites the previous value and thereby permanently changes the reaction's priority.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance of the reaction for which the priority is to be set
`Reaction`	non-negative integer, which is the index into `TReactionsInfo` for the reaction for which the priority is to be set
`Priority`	non-negative integer of the reaction's priority

Return Values

`TRUE`	`SetReactionPriority()` was able to set the priority for the reaction.
`FALSE`	`SetReactionPriority()` failed to set the priority for the reaction.

Errors

The function will fail if

- the reaction index is less than 0 or greater than or equal to the number of reactions supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed
- `Priority` is less than `RT_PRIORITY_MIN` or greater than `RT_PRIORITY_MAX`

See Also

`CreateInstance()`

Example

```
if (!SetReactionPriority
  (InstanceSimpleClock1,0,CLOCK_PRIORITY))
{
  RtPrintf
   ("failed to set reaction 0 measurement for SimpleClock1\n");
}
```

SetReactionQueueLength

Synopsis

```
#include <PinInterface.h>
BOOL SetReactionQueueLength (
   TComponentInstance *Instance,
   unsigned int Reaction,
   short QueueLength
);
```

Description

`SetReactionQueueLength()` is used to set the queue length of the message queue for a component instance's reaction. The new value overwrites the previous value and thereby permanently changes the queue's length.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance of the reaction for which the queue size is to be set
`Reaction`	non-negative integer which is the index into `TReactionsInfo` for the reaction for which the queue size is to be set
`QueueLength`	non-negative integer of the queue size for holding messages for the reaction

Return Values

`TRUE`	`SetReactionQueueLength()` was able to set the queue size for the reaction.
`FALSE`	`SetReactionQueueLength()` failed to set the queue size for the reaction.

Errors

The function will fail if

- the reaction index is less than 0 or greater than or equal to the number of reactions supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed
- `QueueLength` is less than 0

See Also

`CreateInstance()`

Example

```
if (!SetReactionQueueLength(InstanceSimpleClock1, 0, 1))
{
  RtPrintf
    ("failed to set reaction 0 measurement for SimpleClock1\n");
}
```

SetReactionTimeOut

Synopsis

```
#include <PinInterface.h>
BOOL SetReactionTimeOut (
   TComponentInstance *Instance,
   unsigned int Reaction,
   long Timeout
);
```

Description

`SetReactionTimeOut()` is used to set the timeout period of a component instance's reaction. The value set overwrites the previous value and thereby permanently changes the reaction's timeout.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance of the reaction for which the timeout is to be set
`Reaction`	non-negative integer, which is the index into `TReactionsInfo` for the reaction for which the timeout is to be set
`TimeOut`	non-negative integer indicating the reaction's timeout that will generate a `TIMEOUT` message should no other message type arrive

Return Values

`TRUE`	`SetReactionTimeOut()` was able to set the timeout for the reaction.
`FALSE`	`SetReactionTimeOut()` failed to set the timeout for the reaction.

Errors

The function will fail if

- the reaction index is less than 0 or greater than or equal to the number of reactions supported by the component
- an invalid pointer, or a pointer initialized to something other than that returned by `LoadComponent()`, was passed

See Also

`CreateInstance()`

Example

```
if (!SetReactionTimeOut(InstanceSimpleClock1,0,clock1Period))
{
  RtPrintf
    ("failed to set reaction 0 measurement for SimpleClock1\n");
}
```

SourceAddSinkPin

Synopsis

```
#include <PinInterface.h>
BOOL SourceAddSinkPin (
   TComponentInstance *Instance,
   unsigned int SourcePin,
   char *SinkComponentUniqueName,
   unsigned int SinkPin
);
```

Description

SourceAddSinkPin() is used to dynamically establish an interaction between two components loaded by the controller. A sink pin is added to a source pins interaction list via the supplied parameters.

Instance	non-null pointer returned by the CreateInstance() of the component instance of the source pin to be connected
SourcePin	index of the source pin to be connected
SinkComponentUniqueName	non-null, non-empty character string (null-terminated), which is unique to the component instance of the sink pin to which the source pin is to be connected
SinkPin	index of the sink pin to which the source pin should be connected

Return Values

TRUE	SourceAddSinkPin() was able to connect the designated source pin to the designated sink pin.
FALSE	SourceAddSinkPin() failed to connect the designated source pin to the designated sink pin.

Errors

`SourceAddSinkPin()` will fail if

- the designated source pin or sink pin is invalid
- either instance of the components is not properly created with `CreateInstance()`
- there is a memory allocation error

See Also

`CreateInstance()`

Example

```
if (SourceAddSinkPin
  (InstanceSimpleClock1, 0,
   EMoveInstance1->UniqueName , 1) == FALSE) {
    RtPrintf
      ("SourceAddSinkPin clock1.r0 ->> EMove.s1 Failed \n");
   } else {
    RtPrintf
      ("clock1.r0 Source ->> EMove.s1 Sink OK!\n");
   }
```

`StartInstance`

Synopsis

```
#include <PinInterface.h>
BOOL StartInstance (
   TComponentInstance *Instance
);
```

Description

`StartInstance()` is used to start an instance of a component, which signifies that messages sent to the component instance will cause the associated reactions to be triggered. An instance of a component that has not been started will not react to messages sent to sink pins and reactions will not execute.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance to be started

Return Values

`TRUE`	`StartInstance()` successfully started the component instance.
`FALSE`	`StartInstance()` failed to successfully start the component instance.

Errors

`StartInstance()` will fail if the underlying system is unable to successfully create and start a separate thread of control to manage reactions and messages for the component instance.

See Also

`CreateInstance(), StopInstance()`

Example

```
if (!StartInstance(InstanceSimpleClock1)) {
  RtPrintf ("StartInstance clock1 Failed\n");
} else {
  RtPrintf ("StartInstance clock1 Ok!\n");
}
```

`StartPinInterface`

Synopsis

```
#include <PinInterface.h>
BOOL StartPinInterface (void);
```

Description

`StartPinInterface()` is called to initialize and start the Pin component runtime. This function must be called prior to using any other functions supported by the Pin runtime.

Return Values

`TRUE`	`StartPinInterface()` successfully started the Pin runtime.
`FALSE`	`StartPinInterface()` failed to successfully start the Pin runtime.

Errors

`StartPinInterface()` will fail if it is unable to

- allocate shared memory for the interprocess communication (needed for `SendOut-SourcePin()`)
- link to the Pin directory server (needed for `CreateInstance()` and `SourceAddSinkPin()`)
- load and initialize the dynamically linked IPC mechanism

See Also

`StopPinInterface()`

Example

```
if (!StartPinInterface()) {
  RtPrintf("Failed to start the Pin Interface\n");
  ExitProcess(0);
}
else
  RtPrintf("Pin Interface Successfully Started\n");
```

StopInstance

Synopsis

```
#include <PinInterface.h>
BOOL StopInstance (
   TComponentInstance *Instance
);
```

Description

`StopInstance()` is used to shut down an instance of a component. After being shut down, the component instance will no longer react to messages sent to sink pins, and all reaction handlers stop.

`Instance`	non-null pointer returned by the `CreateInstance()` of the component instance to be stopped

Return Values

`TRUE`	`StopInstance()` successfully stopped the component instance.
`FALSE`	`StopInstance()` failed to successfully stop the component instance.

Errors

None

See Also

`CreateInstance()`, `StartInstance()`

Example

```
StopInstance (InstanceSimpleClock1);
```

`StopPinInterface`

Synopsis

```
#include <PinInterface.h>
BOOL StopPinInterface (void);
```

Description

`StopPinInterface()` is called to gracefully shut down the Pin component runtime. This function should be the last one called prior to ending or exiting the controller.

Return Values

TRUE	`StopPinInterface()` successfully stopped the Pin runtime.
FALSE	`StopPinInterface()` failed to successfully stop the Pin runtime.

Errors

`StopPinInterface()` will fail if it is unable to successfully unload the IPC mechanism and free previously allocated memory.

See Also

`StartPinInterface()`

Example

```
if (!StopPinInterface()) {
  RtPrintf("Failed to stop the Pin Interface\n");
  ExitProcess(0);
}
else
  RtPrintf("Pin Interface Successfully Stopped\n");
```

UnloadComponent

Synopsis

```
#include <PinInterface.h>
BOOL UnloadComponent (
   TPinComponent *PinComponent
);
```

Description

`UnloadComponent()` is used to dynamically remove a component from the controller's memory space. Once a component is removed, all instances of the component will become invalid; therefore, it is critical that all component instances be deleted using `DeleteInstance()` prior to unloading the component.

`Component`	non-null pointer for a component whose address space was attached by `LoadComponent()`

Return Values

`TRUE`	`UnloadComponent()` successfully unloaded the Pin component.
`FALSE`	`UnloadComponent()` failed to successfully unload the Pin component.

Errors

`UnloadComponent()` will fail if

- the component was not previously loaded
- the pointer to the component is invalid or was not returned by `LoadComponent()`
- the underlying system mechanism for unallocating the dynamic library of the component fails (which in the case of Win32 is `FreeLibrary()`)

See Also

`LoadComponent()`, `FreeLibrary()` for the Win32 API

Example

```
if (!UnloadComponent(PinSimpleClock)) {
  RtPrintf("Failed to Unload SimpleClock\n");
  ExitProcess(0);
}
else
  RtPrintf("SimpleClock Unload Successful\n");
```

`WaitForNotifications`

Synopsis

```
#include <PinInterface.h>
int WaitForNotifications (
   TControllerMsg *msg,
   DWORD dwMilliseconds
);
```

Description

`WaitForNotifications()` is used by a controller to wait in a blocking or non-blocking mode for (1) a user-defined message or (2) code and an optional character string from a component instance. This function is useful in communicating exception conditions to the controller, requiring controller attention (such as shutdown).

`msg`	non-null pointer to a `TControllerMsg` structure having user-defined semantics
`dwMilliseconds`	length of time in milliseconds to wait for a notification before returning with `NOTIFY_TIMEOUT`. A timeout of `IPC_WAITFOREVER` will cause `WaitForNotifications()` to block until such time that a notification message is sent to the assembly controller.

Return Values

`SUCCESS`	The notification message has been received successfully.
`CONTROLLER_NOT_FOUND`	Either the shared memory segment for message passing has not been initialized (via `StartPinInterface()`), or the queue for delivering messages to the controller has been deleted.
`INVALID_MSG_POINTER`	msg was passed a NULL pointer.

CONTROLLER_QUEUE_ERROR	There was an error in processing the notification queue on the receipt of a message.
NOTIFY_TIMEOUT	The timeout specified in dwmilliseconds expired.
CONTROLLER_UNKNOWN_ERROR	The underlying IPC mechanism for delivering messages failed for an unknown reason.

Errors

The errors generated by this function and the conditions that cause them are described above under "Description."

See Also

NotifyController()

Example

```
do {
  retval=WaitForNotifications(&CMsg,5000);
  if ( retval==SUCCESS) {
    notifycount++;
    RtPrintf("Received Notification From %s %d %s\n",
      CMsg.Instance, CMsg.Code, CMsg.Message);
    if (notifycount==1)
    {
      RtPrintf("Received 1st notify - terminating program\n");
       break;
    }
  }
  else if (retval==NOTIFY_TIMEOUT) {
   RtPrintf("Notification Timeout\n");
  }
  else
    printf("Notify Error\n");
} while((retval==NOTIFY_TIMEOUT) || ( retval==SUCCESS));
```

Appendix B Example

In this appendix, we present a significant fragment of a real (albeit toy) application in Pin. Our objective is to provide the interested reader with an example of the use of the Pin API. The logical structure of the application is first presented using the iconography and formal syntax of CCL [Wallnau 03a]. Then, the generated code for a component implementation and assembly controller are presented. The generated code is presented "as generated" with minimal reformatting.

B.1 A Simple Assembly

In this section, we present the specification for a trivial application consisting of three components: (1) one that takes input from the keyboard, (2) one that puts output to the display, and (3) a component that maintains a buffer of length 1 of keyboard input.

The assembly is depicted in Figure 7.

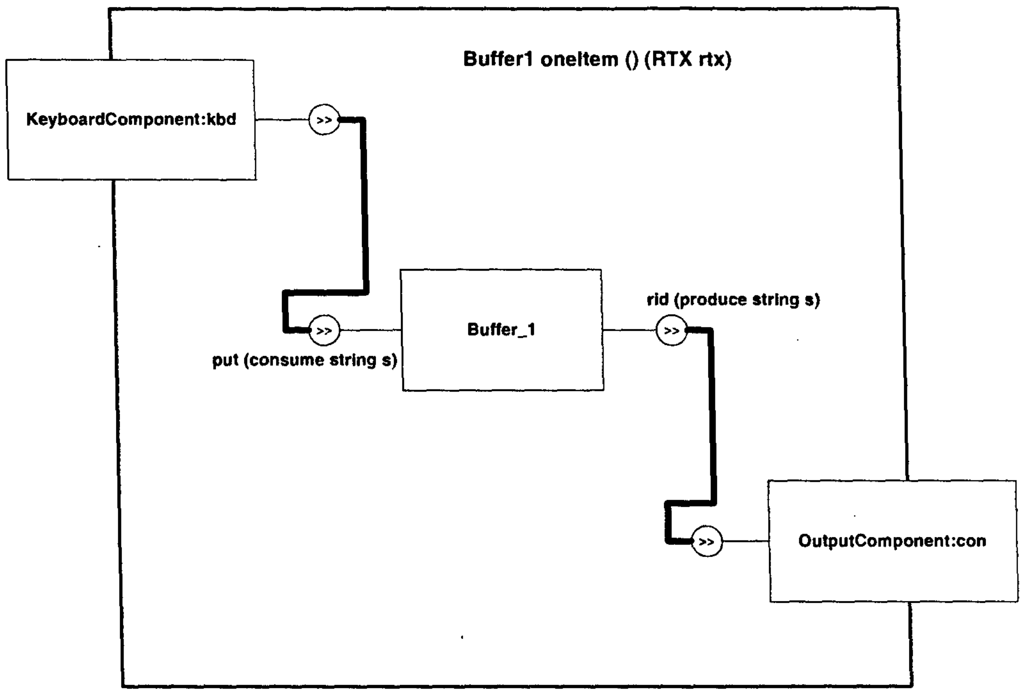


Figure 7: A Simple Assembly Specification

The behavior of the `Buffer_1` component is trivial; a state machine for the behavior of the only reaction in `Buffer_1` is shown in Figure 8. The accepting state is `listen`; in this state, the reaction is prepared to accept stimulus on the `put` sink pin. When the buffer is full (in this case, when there is one item in the buffer), the previously buffered item is forwarded to the `rid` source pin.

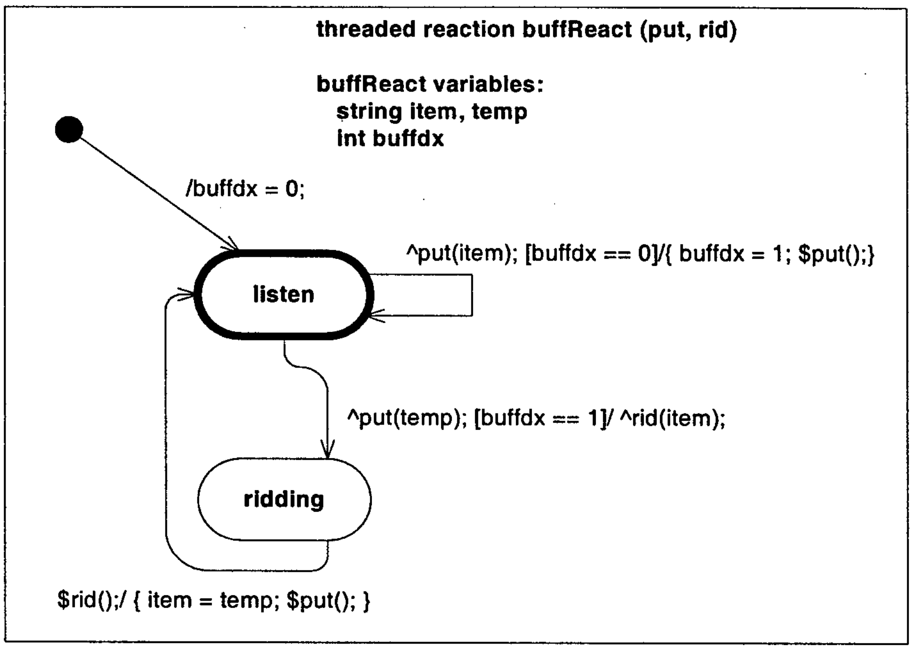

threaded reaction buffReact (put, rid)
buffReact variables:
string item, temp
int buffdx
/buffdx = 0;
^put(item); [buffdx == 0]/{ buffdx = 1; $put();}
listen
^put(temp); [buffdx == 1]/ ^rid(item);
ridding
$rid();/ { item = temp; $put(); }

Figure 8: Statechart for Buffer_1 Reaction

The CCL specification for this simple application is provided below. Keywords are shown in **boldface**.

```
component Buffer_1 () {
   sink asynch put(consume string s);
   source unicast rid (produce string s);

   threaded react buffReact (put, rid) {
      string item, temp;
      int buffdx;

      start -> listen {
         action buffdx = 0;
      }

      listen -> listen {
         trigger ^put(item);
         guard buffdx == 0;
         action {
            buffdx = 1;
            $put();
         }
      }

      listen -> ridding {
         trigger ^put(temp);
         guard buffdx == 1;
         action ^rid(item);
      }

      ridding -> listen {
         trigger $rid();
         action {
            item = temp;
            $put();
         }
      }
   }
}
```

The CCL specification for the environment specifies which environment-provided services a component may use. While services are implemented as components, some rules on what services are allowed to do are more relaxed than those for components. The motivation for this condition is beyond the scope of this report. The specification of the `KeyboardComponent` and `OutputComponent` components are placeholders for more complex specifications and are not included here.

```
environment RTX()
{
   singleton service KeyboardComponent () {
      source unicast putKeyboard (produce string msg);
      threaded react eternal_TBD (putKeyboard) {
         start -> putting { }
         putting->putting { }
      }
   }

   service OutputComponent ()
   {
      sink asynch putConsole (consume string msg);
      threaded react eternal_TBD (putConsole) {
         start -> writing { }
         writing -> writing { }
      }
   }
}// RTX
```

The specification of the Buffer_1 component is provided below. The correspondence between the syntax of CCL reactions and the graphical (UML-like) statechart shown in Figure 8 is straightforward.

The trickiest part of the specification lies in the instantiation of assemblies and environments. First, we create an instance of the runtime environment (RTX), called `env` in the example. Next, we create an instance of the assembly `Buffered` (called `simpleBuffered`) and use the services provided by `env` to satisfy the assumptions of the `Buffered` assembly type.

```
// specify the assembly as a topology of component instances
Assembly Buffered () (RTX)
{
   // what we require of the environment is an "assumption"
   assume {
      RTX:KeyboardComponent keyb();
      RTX:OutputComponent outp();
   }

   Buffer_1 buff();

   // here is the wiring
   keyb:putKeyboard~> buff:put;
   buff:rid~> outp:putConsole;

   expose {}
}

// instantiate the RTX runtime environment
RTX env() {
   RTX:KeyboardComponent kb();
   RTX:OutputComponent cns();
};

// instantiate the assembly in the env instance of RTX
Buffered simpleBuffered() {
   Buffered:keyb = env:kb;
   Buffered:outp = env:cns;
};
```

B.2 Component Buffer_1 Implementation (Generated)

The CCL specifications in the previous section are sufficient to generate a working implementation of the `Buffer_1` component and the `simpleBuffered` assembly.

The code below is the generated code for the custom part of the `Buffer_1` component. The code for the reaction handler that implements the `Buffer_1` state machine is shown in **boldface**. The code is shown as generated, without additional formatting.

```
//
// Functions called from Buffer_1.c
//
#include <windows.h>
#include <rtapi.h>
#include "ComponentSpec.h"
#include "Printf.h"
#include "Libc_Support.h"
#include "ComponentFuncs.h"
#include "ComponentArgs.h"

typedef char *STRING;

typedef struct _COMPONENT_Buffer_1_VARS {
   void *dummy; // always at least this member
} COMPONENT_Buffer_1_VARS;

typedef struct _REACTION_buffReact_VARS {
   STRING temp;
   STRING item;
   int buffdx;
   int CURRENT_STATE;
} REACTION_buffReact_VARS;

typedef struct __SOURCE_PIN_rid_PRODUCE_PARAMS {
   STRING s;
} SOURCE_PIN_rid_PRODUCE_PARAMS;

typedef struct __SINK_PIN_put_CONSUME_PARAMS {
   STRING s;
} SINK_PIN_put_CONSUME_PARAMS;

typedef struct _INSTANCE_DATA{
   COMPONENT_Buffer_1_ARGS *Buffer_1_args;
   COMPONENT_Buffer_1_VARS *Buffer_1_vars;
   REACTION_buffReact_VARS *buffReact_vars;
   SINK_PIN_put_CONSUME_PARAMS *put_params;
} INSTANCE_DATA;


//-------------- ***REACTION DECLARATIONS FOR buffReact -------
--------
```

```
//-------------- Timeout Handler for buffReact
long REACTION_buffReact_TIMED_EVENT_HANDLER(TReactions *Reac-
tion, TReason Reason, void *Data);


//-------------- Reaction Main Event Handler for buffReact
BOOL REACTION_buffReact_PIN_GENERAL_EVENT_HANDER(TReactions
*Reaction, TReason Reason, void *Data);


//-------------- Reaction Sink Pin Array Declaration for buf-
fReact ---------------
unsigned int REACTION_buffReact_SINK_ARRAY [
REACTION_buffReact_NUM_SINKS ] = {0 /* put */};


//-------------- Reaction Source Pin Array Declaration for buf-
fReact ---------------
unsigned int REACTION_buffReact_SOURCE_ARRAY [
REACTION_buffReact_NUM_SOURCES ] = {0 /* rid */};


//-------------- Reaction State Machine Info Array for buffRe-
act
static int REACTION_buffReact_ACCEPTING_STATES [] = {0, 1, 0};

static int REACTION_buffReact_ACCEPTS_INTERACTION [3][1] = {
   {0},
   {1},
   {0}};


//-------------- ***COMPONENT DECLARATIONS ---------------


//-------------- Index array for sink pins ---------------
TPinInfoSink SinkPins [NUM_SINK_PINS] = {
   { "(put)", "TEST"}
};


//-------------- Index array for source pins ---------------
TPinInfoSource SourcePins [NUM_SOURCE_PINS] = {
   { "(rid)", ""}
```

```
};


//-------------- Reaction array for component --------------
TReactionsInfo ReactionInfo[ 1 ] = {
      {REACTION_buffReact_NUM_SOURCES,//number of source pins
      REACTION_buffReact_NUM_SINKS,// number of sink pins
      REACTION_buffReact_SOURCE_ARRAY,//ordered array of source
pin numbers
      REACTION_buffReact_SINK_ARRAY,//ordered array of sink pin
numbers
      DEFAULT_QUEUE_SIZE,
      DEFAULT_PRIORITY,
      IPC_WAITFOREVER,// default timeout
      FALSE,      // default measurement flag
      REACTION_buffReact_PIN_GENERAL_EVENT_HANDER, // for pin
*and* timed events
      REACTION_buffReact_TIMED_EVENT_HANDLER// for UML timed
events
   }
};



//-------------- Component Info Struct --------------
TComponentInfo ComponentInfo = {
   "Buffer_1",
   NUM_SOURCE_PINS,
   NUM_SINK_PINS,
   NUM_REACTIONS};

//-------------- Life Cycle Operations --------------

//-------------- Create Component ---------------------
BOOL CreateComponentInstance(void **Data, void *State, unsigned
int SizeOfState)
{
   if (Data==NULL) {
      return (FALSE);
   }
   *Data = NULL;
   if (State != NULL) {
      *Data = RtAllocateLockedMemory( sizeof (INSTANCE_DATA )
);
      ((INSTANCE_DATA*)*Data)->Buffer_1_args = RtAllocate-
LockedMemory( SizeOfState );
      Rt_memcpy(((INSTANCE_DATA *)*Data)->Buffer_1_args, State,
SizeOfState);
```

```
      ((INSTANCE_DATA *)*Data)->Buffer_1_vars = RtAllocate-
LockedMemory(sizeof (COMPONENT_Buffer_1_VARS));

      // --- now allocate static variables for each reaction ---

      ((INSTANCE_DATA *)*Data)->buffReact_vars = RtAllocate-
LockedMemory(sizeof (REACTION_buffReact_VARS));
      ((INSTANCE_DATA *)*Data)->buffReact_vars->CURRENT_STATE =
0;

      // --- now allocate space for the consume parameters of
source pins ---


      // --- now allocate space for the consume parameters of
sink pins ---

      ((INSTANCE_DATA *)*Data)->put_params = RtAllocateLocked-
Memory(sizeof (SINK_PIN_put_CONSUME_PARAMS));


      // --- initialize component local variables if necessary -
--


      // --- initialize reaction local variables if necessary -
--

   }

   return (TRUE);
}

//------------- Delete Component ---------------------
BOOL DeleteComponentInstance(void **Data)
{
   if (Data==NULL) {
      return(FALSE);
   }
   else {

      // --- free component-level resources ------------------
------
      RtFreeLockedMemory(((INSTANCE_DATA *)*Data)-
>Buffer_1_args);
      RtFreeLockedMemory(((INSTANCE_DATA *)*Data)-
>Buffer_1_vars);
```

```
     // --- free component-level resources ------------------
------
     RtFreeLockedMemory( ((INSTANCE_DATA *)*Data)-
>buffReact_vars );

     // --- free source pin having consume parameters --------
----

     // --- and now sink pins having consume parameters ------
----
     RtFreeLockedMemory( ((INSTANCE_DATA *)*Data)->put_params
);
     RtFreeLockedMemory(*Data);
     return (TRUE);
   }
}

//------------- Reaction Initializers  ---------------------
void ReactionInitialize(TReactions *Reaction, int ReactionIn-
dex) {
   // nothing, for now
}

//------------- Reaction Terminator  ---------------------
void ReactionTerminating(TReactions *Reaction, int ReactionIn-
dex) {
   // nothing, for now
}


//------------- Functions for Reaction buffReact ------------
--
long REACTION_buffReact_TIMED_EVENT_HANDLER(TReactions *Reac-
tion, TReason Reason, void *Data)
{
   // not currently used
   return (0);
}


//------------- Reaction Main Event Handler for buffReact
BOOL REACTION_buffReact_PIN_GENERAL_EVENT_HANDER(TReactions
*Reaction, TReason Reason, void *Data)
{

   // pre-defined variables...
   IPC_MSG MessageOut;
   int __marshDx;
   char *__marshString;
   INSTANCE_DATA *p;
   int CURRENT_STATE;

   int cmd;
   TCommonMsg *msg;
   IPC_MSG *MessageIn;


   // user-defined variables...

   //--------------------declare component or service state
variables----------

   //--------------------declare reaction state variables------
----------------
   STRING buffReact_temp;
   STRING buffReact_item;
   int buffReact_buffdx;


   if (Reason == TIMEOUT) { return (TRUE); } // TBD

   if (Reason == TMR) { return (TRUE); } // TBD

   // Reason == PINMSG

   msg = (TCommonMsg *)Data;
   MessageIn = msg->MessageIn;
   cmd = msg->cmd;
   p = Reaction->Instance->InstanceData;
   CURRENT_STATE = p->buffReact_vars->CURRENT_STATE;

   buffReact_temp = p->buffReact_vars->temp;
   buffReact_item = p->buffReact_vars->item;
   buffReact_buffdx = p->buffReact_vars->buffdx;

   if (CURRENT_STATE == 0) {

      // START->listen transition action:
      buffReact_buffdx =  0 ;
      // no listen entry action
      CURRENT_STATE = 1;
   }
```

```
//------------- state machine for buffReact --------------

do {
   switch(CURRENT_STATE) {
   case 1: // listen
      // no listen exit action
      if (cmd == 0) {

         // listen->listen guard
         if ( buffReact_buffdx ==  0  ) {

            // Unmarshall params for listen->listen
            __marshDx = 0;
            //--- STRING (char *) ---

            buffReact_item = RtAllocateLockedMemory(
strlen((char *) &MessageIn->data[__marshDx]) + 1);
            Rt_strcpy(buffReact_item, (char *) &MessageIn-
>data[__marshDx]);
            __marshDx += strlen((char *) &MessageIn-
>data[__marshDx]) + 1;

            // listen->listen action:
            {
               buffReact_buffdx =  1 ;

               // called on an asynchronous pin -- no data to
be returned to caller --

            }

            // no listen entry action

            CURRENT_STATE = 1;
            p->buffReact_vars->CURRENT_STATE = CURRENT_STATE;
            p->buffReact_vars->temp = buffReact_temp;
            p->buffReact_vars->item = buffReact_item;
            p->buffReact_vars->buffdx = buffReact_buffdx;
         }

         // listen->ridding guard
         else if ( buffReact_buffdx ==  1  ) {

            // Unmarshall params for listen->ridding
            __marshDx = 0;
            //--- STRING (char *) ---
```

```
                buffReact_temp = RtAllocateLockedMemory(
strlen((char *) &MessageIn->data[__marshDx]) + 1);
                Rt_strcpy(buffReact_temp, (char *) &MessageIn-
>data[__marshDx]);
                __marshDx += strlen((char *) &MessageIn-
>data[__marshDx]) + 1;

                // listen->ridding action:

                //-----------------Marshall each actual param on
interaction in rid_outboundParams---------------
                __marshDx = 0;

                //----- string (char) * s ------//
                __marshString = buffReact_item;
                Rt_strcpy(&MessageOut.data[__marshDx],
__marshString);
                __marshDx += Rt_strlen (__marshString) + 1;
                //------------------Call asynchronous IPC mecha-
nism ----------------
                if (!SendOutSourcePin(
                    Reaction, 0, &MessageOut, (short)
(sizeof(MessageOut.data)), IPC_WAITFOREVER /* TBD property */))
{
                   NotifyContoller(Reaction->Instance,
CONTROLLER_UNKNOWN_ERROR, "error in SendOutSourcePin");
                   return(FALSE);
                }
                //------------------------------------------------
------------------


                // no ridding entry action

                CURRENT_STATE = 2;
             }
             else {
                // throw it away--no satisfied guard
             }
          }
          break;
       case 2: // ridding
          // no ridding exit action
          //-----------------Retrieve callback data from rid
into local state --------

          //------------------------------------------------
----------------------
```

```
         // ridding->listen action:
         {
            buffReact_item = buffReact_temp;

            // called on an asynchronous pin -- no data to be
returned to caller --

         }

         // no listen entry action

         CURRENT_STATE = 1;
         p->buffReact_vars->CURRENT_STATE = CURRENT_STATE;
         p->buffReact_vars->temp = buffReact_temp;
         p->buffReact_vars->item = buffReact_item;
         p->buffReact_vars->buffdx = buffReact_buffdx;

         break;
      default:
         NotifyContoller(Reaction->Instance,
CONTROLLER_UNKNOWN_ERROR, "Unrecognized state");
         return(FALSE);
      }
   } while (!REACTION_buffReact_ACCEPTING_STATES[ CURRENT_STATE
]);

   return (TRUE);
}
```

B.3 Controller Implementation (Generated)

The code shown below appears as generated, with the exception that some extraneous empty lines have been removed. The correspondence between the generated code and the assembly life cycle depicted in Figure 4 are obvious from the generated comments.

```
// simpleBuffered.c
#include "simpleBuffered.h"
#include "PinInterface.h"
#include "PinDirectoryServer.h"
#include "Printf.h"
#include "Libc_Support.h"

// the include files for any parameterized components/services
#include "../KeyboardComponent/ComponentArgs.h"
#include "../Buffer_1/ComponentArgs.h"
```

```
#include "../OutputComponent/ComponentArgs.h"


void _cdecl main(int argc, char **argv, char **envp)
{
   TPinComponent*factories[3];
   TComponentInstance*instances[3];
   HANDLE ThreadHandle;
   INT Priority;
   TControllerMsg CMsg;
   int retVal;

   COMPONENT_KeyboardComponent_ARGS KeyboardComponent_args;
   COMPONENT_Buffer_1_ARGS Buffer_1_args;
   COMPONENT_OutputComponent_ARGS OutputComponent_args;

// start up pin interface...

   if (!StartPinInterface()) {
      RtPrintf("Failed to start the Pin Interface\n");
      ExitProcess(0);
   }


// load components used in this assembly...

   factories[0] = LoadComponent("KeyboardComponent.dll");
   if (factories[0] == NULL) {
      RtPrintf("Failed to Load KeyboardComponent\n");
      ExitProcess(0);
   }
   else {
      RtPrintf("KeyboardComponent Load Successful\n");
   }

   factories[1] = LoadComponent("Buffer_1.dll");
   if (factories[1] == NULL) {
      RtPrintf("Failed to Load Buffer_1\n");
      ExitProcess(0);
   }
   else {
      RtPrintf("Buffer_1 Load Successful\n");
   }

   factories[2] = LoadComponent("OutputComponent.dll");
   if (factories[2] == NULL) {
      RtPrintf("Failed to Load OutputComponent\n");
      ExitProcess(0);
   }
   else {
      RtPrintf("OutputComponent Load Successful\n");
   }

// instantiate components and services used in the assembly


   if ((instances[0] = CreateInstance(factories[0], "keyb",
&KeyboardComponent_args, sizeof (KeyboardComponent_args) )) !=
NULL) {
      RtPrintf("keyb instantiated\n");
   } else {
      RtPrintf("keyb FAILED TO BE instantiated\n");
   }

   if ((instances[1] = CreateInstance(factories[1], "buff",
&Buffer_1_args, sizeof (Buffer_1_args) )) != NULL) {
      RtPrintf("buff instantiated\n");
   } else {
      RtPrintf("buff FAILED TO BE instantiated\n");
   }

   if ((instances[2] = CreateInstance(factories[2], "outp",
&OutputComponent_args, sizeof (OutputComponent_args) )) !=
NULL) {
      RtPrintf("outp instantiated\n");
   } else {
      RtPrintf("outp FAILED TO BE instantiated\n");
   }

// initialize properties of reactions

   //set properties of reaction eternal_TBD of instance keyb
   //set properties of reaction buffReact of instance buff
   //set properties of reaction eternal_TBD of instance outp


// configure (or initialize) each component/service instance

   if (ConfigureInstance(instances[0]) == FALSE) {
      RtPrintf("Configure keyb FAILED\n");
   }

   if (ConfigureInstance(instances[1]) == FALSE) {
      RtPrintf("Configure buff FAILED\n");
   }
   if (ConfigureInstance(instances[2]) == FALSE) {
      RtPrintf("Configure outp FAILED\n");
   }


// wire the assembly together

   if (!((instances[0]) && (instances[1]) && (instances[2])))
   {
      RtPrintf("Not all instances created -- Skipped Wir-
ing\n");
      //should clean-up and exit?
   }
   else {
      if (SourceAddSinkPin(instances[0], 0, instances[1]-
>UniqueName, 0) == FALSE) {
         RtPrintf("SourceAddSinkPin keyb:putKeyboard ->
buff:put Failed\n");
      }

      if (SourceAddSinkPin(instances[1], 0, instances[2]-
>UniqueName, 0) == FALSE) {
         RtPrintf("SourceAddSinkPin buff:rid -> outp:putConsole
Failed\n");
      }

   }

   ThreadHandle = GetCurrentThread();
   Priority = RtGetThreadPriority(ThreadHandle);
   RtSetThreadPriority(ThreadHandle, RT_PRIORITY_MAX);
   RtSetThreadTimeQuantum(ThreadHandle, 0);
   RtPrintf("Thread Priority is %d\n", RtGetThreadPrior-
ity(ThreadHandle));

// start component instances

   StartInstance(instances[1]);

// start service instances

   StartInstance(instances[0]);
   StartInstance(instances[2]);

// start clock instances (last to start)

// wait for asembly to be terminated...
```

```
   retVal = WaitForNotifications(&CMsg, IPC_WAITFOREVER);
   switch(retVal) {
      case SUCCESS:
         break;
      case NOTIFY_TIMEOUT:
         break;
      default: // some kind of error occurred
         RtPrintf("Error: %s : %s\n", CMsg.Instance, CMsg.Mes-
sage);
         break;
   }

// shutdown clock instances (first to stop)

// shutdown service instances
   StopInstance(instances[0]);
   StopInstance(instances[2]);

// shutdown component instances

   StopInstance(instances[1]);
// restore the assembly (main) priority to prior value (see
above)
   RtSetThreadPriority(ThreadHandle, Priority);

// destroy component and service instances
   if (instances[0]) DeleteInstance(instances[0]);
   if (instances[1]) DeleteInstance(instances[1]);
   if (instances[2]) DeleteInstance(instances[2]);

// unload components used in this assembly...
   if(!UnloadComponent(factories[0])) {
      RtPrintf("Failed to unload KeyboardComponent\n");
   }
   if(!UnloadComponent(factories[1])) {
      RtPrintf("Failed to unload Buffer_1\n");
   }

   if(!UnloadComponent(factories[2])) {
      RtPrintf("Failed to unload OutputComponent\n");
   }

// shutdown pin interface
   if(!StopPinInterface()) {
      RtPrintf("Failed to stop the Pin Interface\n");
   }
   ExitProcess(0);
}
```

References

[Bachmann 00]	Bachmann, F.; Bass, L.; Buhman, C.; Comella-Dorda, S.; Long, F.; Robert, J.; Seacord, R.; & Wallnau, K. *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition* (CMU/SEI-2000-TR-008, ADA379930). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/publications/documents /00.reports/00tr008.html
[Bass 05]	Bass, L.; Ivers, J.; Klein, M.; & Merson, P. *Reasoning Frameworks* (CMU/SEI-2005-TR-007). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports /05tr007.html
[Hissam 04a]	Hissam, S.; & Klein, M. *A Model Problem for an Open Robotics Controller* (CMU/SEI-2004-TN-030). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports /04tn030.html

[Ivers 02]	Ivers, J.; Sinha, N.; & Wallnau, K. *A Basis for Composition Language CL* (CMU/SEI-2002-TN-026, ADA407797). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tn026.html
[Plakosh 99]	Plakosh, D.; Smith, D.; & Wallnau, K. *Builder's Guide for Water-Beans Components* (CMU/SEI-99-TR-024, ADA373154). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. http://www.sei.cmu.edu/publications/documents/99.reports/99tr024/99tr024abstract.html
[Szyperski 02]	Szyperski, C.; Gruntz, D.; & Murer, S. *Component Software: Beyond Object-Oriented Programming, Second Edition*. Boston, MA: Addison-Wesley, 2002.
[Wallnau 03a]	Wallnau, K. & Ivers, J. *Snapshot of CCL: A Language for Predictable Assembly* (CMU/SEI-2003-TN-025, ADA418453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tn025.html
[Wallnau 03b]	Wallnau, K. *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tr009.html
[Ward 00]	Ward-Dutton, N. Containers: "A Sign Components are Growing Up." *Application Development Trends 7*, 1 (January 2000): 41-44, 46.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY** (leave blank)	2. **REPORT DATE** April 2005	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** Pin Component Technology (V1.0) and Its C Interface		5. **FUNDING NUMBERS** FA8721-05-C-0003
6. **AUTHOR(S)** Scott Hissam, James Ivers, Daniel Plakosh, Kurt C. Wallnau		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2005-TN-001
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK, 5 Eglin Street, Hanscom AFB, MA 01731-2116		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. **SUPPLEMENTARY NOTES**		
12.a **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

Pin is a basic, simple component technology suitable for building embedded software applications. Pin implements the container idiom for software components. Containers provide a prefabricated "shell" in which custom code executes and through which all interactions between custom code and its external environment are mediated. Pin is a component technology for pure assembly—systems are assembled by selecting components and connecting their interfaces (which are composed of communication channels called pins).

This report describes the main concepts of Pin and documents the C-language interface to Pin V1.0.

14. **SUBJECT TERMS** component technology, component model, containers, runtime environment, prediction-enabled component technology, embedded systems, application programming interface			15. **NUMBER OF PAGES** 126
			16. **PRICE CODE**
17. **SECURITY CLASSIFICATION OF REPORT** UNCLASSIFIED	18. **SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	19. **SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	20. **LIMITATION OF ABSTRACT** UL